


```

DDDDDDDD  BBBB BBBB  GGGGGGGG  LL  IIIIII  BBBB BBBB
DDDDDDDD  BBBB BBBB  GGGGGGGG  LL  IIIIII  BBBB BBBB
DD  DD  BB  BB  GG  LL  II  BB  BB
DD  DD  BB  BB  GG  LL  II  BB  BB
DD  DD  BB  BB  GG  LL  II  BB  BB
DD  DD  BBBB BBBB  GG  LL  II  BBBB BBBB
DD  DD  BBBB BBBB  GG  LL  II  BBBB BBBB
DD  DD  BB  BB  GG  LL  II  BB  BB
DD  DD  BB  BB  GG  LL  II  BB  BB
DD  DD  BB  BB  GG  LL  II  BB  BB
DD  DD  BB  BB  GG  LL  II  BB  BB
DDDDDDDD  BBBB BBBB  GGGGGG  LLLLLLLLLL  IIIIII  BBBB BBBB
DDDDDDDD  BBBB BBBB  GGGGGG  LLLLLLLLLL  IIIIII  BBBB BBBB

LL  IIIIII  SSSSSSSS
LL  IIIIII  SSSSSSSS
LL  II  SS
LL  II  SS
LL  II  SS
LL  II  SS
LL  II  SSSSSS
LL  II  SSSSSS
LL  II  SS
LL  II  SS
LL  II  SS
LL  II  SS
LLLLLLLLLL  IIIIII  SSSSSSSS
LLLLLLLLLL  IIIIII  SSSSSSSS

```


M 11
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 1
(1)

DBGLIB -- COMMON DEFINITION FILE FOR THE VAX DEBUGGER

Version: 'V04-000'

COPYRIGHT (c) 1978, 1980, 1982, 1984 BY
DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.
ALL RIGHTS RESERVED.

THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
TRANSFERRED.

THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
CORPORATION.

DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.

WRITTEN BY
Bert Beander June, 1980.

MODIFIED BY
John Francis
Ping Sager
Rich Title
Vicki Holt
Sid Maxwell

MODULE FUNCTION:
This REQUIRE file contains all literal, macro, and data structure defin-
itions used by new Bliss modules in the Debugger. Old Bliss modules do
not use this file for historical reasons.

TABLE OF CONTENTS

General Debugger Definitions	5
Data Structure Definition and Access	5
Bliss-32 Language Extensions	5
Error Reporting	7
DEBUG Output Macros	8
Miscellaneous Literals	9
Address Descriptor Definitions	15
Address Expression Descriptor Definitions	16
Command Execution Tree Definitions	17
Command Input Stream Definitions	19
Control Flags	21
Define Symbol Table Definitions	22
Error Status Codes	25
Event Table Entry Definitions	26
Event Table Entry for Break or Trace Point	30
Event Table Entry for Threaded Code Event Point	31
Event Table Entry for /READ, /WRITE, /MODIFY	32
Event Table Entry for /CALL, /BRANCH, /INSTRUCTION	33
Event Table Entry for STEP	34
Event Stepping Descriptor	35
Event DO List Descriptor	36
Event WHEN Descriptor	37
Event Page Descriptor	38
Free Memory Management	39
Format of a Free Memory Block	39
Format of an Allocated Memory Block	40
Overall Structure of Memory Pool	42
Temporary Memory Blocks	44
Image File Header Definitions	45
Image Header Symbol Table Descriptor	45
Image File Debug Module Table (DMT)	47
Least Recently Used Module Table Definitions	49
Lexical Scanner Character Table	50
Lexical Token Entry	52
Operand Lexical Token Entry	54
Operator Lexical Token Entry	55
Terminator Lexical Token Entry	59
Linked List Node Definition	61

0106	0	Moved DST Entry Definitions	62
0107	0		
0108	0	Number Scanner State Table	64
0109	0		
0110	0	Operator Information Table	67
0111	0	Operator Routine Table	68
0112	0	Type Graph	76
0113	0		
0114	0	Type Conversion Information Table	77
0115	0	Type Convert Entry	78
0116	0		
0117	0	Pathname Descriptor Definitions	79
0118	0		
0119	0	Permanent Symbol Descriptor Definitions	81
0120	0		
0121	0	Predeclared Identifier Entry	83
0122	0		
0123	0	Primary and Value Descriptor Header	85
0124	0	Primary and Value Descriptor Header for COBOL and PL/I	86
0125	0		
0126	0	Primary Descriptor Definitions	88
0127	0	Primary Descriptor Root Node	89
0128	0	Primary Descriptor Normal Sub-Node	91
0129	0	Primary Descriptor Array Sub-Node	93
0130	0	Primary Descriptor Record Sub-Node	96
0131	0	Primary Descriptor Variant Sub-Node	97
0132	0		
0133	0	Primary Parser State Table	99
0134	0		
0135	0	Print Information Table	102
0136	0		
0137	0	Register Descriptor Definitions	104
0138	0		
0139	0	Run-Time Symbol Table (RST) Definitions	105
0140	0	RST Entry Common Core	106
0141	0	RST Entry for a Module	109
0142	0	RST Entry for a Routine	112
0143	0	RST Entry for a Lexical Block	114
0144	0	RST Entry for an Entry Point	115
0145	0	RST Entry for an Instruction Label	116
0146	0	RST Entry for a Line Number	117
0147	0	RST Entry for a Data Symbol	118
0148	0	RST Entry for a Data Type Component	120
0149	0	RST Entry for a Data Type	121
0150	0	RST Entry for a Record Variant Set	123
0151	0	RST Entry for an Invocation Number	126
0152	0	RST Entry for an Overloaded Symbol	127
0153	0		
0154	0	Scope List Definitions	128
0155	0		
0156	0	Screen Display Entry Definitions	129
0157	0		
0158	0	Screen Display Line Entry Definitions	133
0159	0	Normal Display Line Entry	133
0160	0	Normal Display Line Entry with Rendition Vector	134
0161	0	Source Display Line Entry	135
0162	0		

C 12
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 4
(2)

0163	0	Screen Pasteboard Entry Definitions	137
0164	00		
0165	00	Screen Window Entry Definitions	139
0166	00		
0167	00	Source Directory Search List	140
0168	00	Source Directory Search List Header Block	141
0169	00	Source Directory Search List Entry	142
0170	00		
0171	00	Source File Control Block	143
0172	00		
0173	00	Source File ID Table	146
0174	00		
0175	00	Source File Record File Address Table	147
0176	00		
0177	00	Static Address Table (SAT) Definitions	148
0178	00		
0179	00	Value Descriptor Definitions	150
0180	00		
0181	00	Old Debugger Definitions	152
0182	0		

GENERAL DEBUGGER DEFINITIONS

The declarations in this section define symbols of general utility throughout the Debugger. This includes widely used literals and various utility macros.

DATA STRUCTURE DEFINITION AND ACCESS

Here we declare all the macros used to define and access BLISS data structures. These are the names L_, W_, B_, W0_, W1_, and so forth. The actual definitions are in a separate REQUIRE file, but they are required for the LIBRARY compilation of the present REQUIRE file.

LIBRARY 'LIB\$:STRUCDEF.L32';

BLISS-32 LANGUAGE EXTENSIONS

Here we declare various minor extensions to the Bliss-32 language.

LITERAL

TRUE = 1, ! Define TRUE and FALSE
FALSE = 0;

!***

MACRO

REPEAT = WHILE 1 DO%, ! Infinite Loop

! ZEROCOR zeroes an area of memory. The two parameters are ADDRESS and COUNT, which are the address of the area to be zeroed and the number of contiguous longwords to be zeroed, respectively.

ZEROCOR (ADDRESS, COUNT) =
CH\$FILL (0, (COUNT) * 4, CH\$PTR (ADDRESS))%,

ONEOF (X) [] =
(MASK (%REMAINING)) ^ (X) LSS 0%,

OR_OP (X) [] =
OR%,

MASK (X) [] =
1 ^ (31-X) OR_OP (%REMAINING) MASK (%REMAINING)%,

CH\$SEQUENCE (N) =

E 12
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 6
(3)

: 0240 0
: 0241 0
: 0242 0

VECTOR [CH\$ALLOCATION (N)]%,
CONSECUTIVE [X] = LITERAL X = %COUNT%;

ERROR REPORTING

The following macro is used for reporting internal DEBUG coding errors symbolically. The macro is used as follows:

```
$DBG_ERROR('MODNAME\ROUTNAME number');
```

This signals the internal DEBUG error message DBGS_INTERR which prints the text of the macro parameter. The convention is that this parameter should include the module name and the routine name of the signal location so that the point where the error was signalled can be easily determined by DEBUG developers. In addition, a number may be included in the signal text to make the text unique when there is more than one \$DBG_ERROR invocation in the same routine.

```
MACRO $DBG_ERROR (STRING) =
  SIGNAL (DBGS_INTERR, 1, UPLIT BYTE (%ASCIC STRING)
  %IF %LENGTH GTR 1
  %THEN
    %REMAINING)
  %ELSE
  )
  %FI %;
```

DEBUG OUTPUT MACROS

The following MACROs are used in several places in the debugger to do formatted ASCII output (\$FAO).

MACRO

! \$FAO_STG_COUNT makes a counted byte string out of an ASCII string. This macro is useful to transform an FAO control string into the address of such a string, whose first byte contains the length of the string in bytes.

\$FAO_STG_COUNT (STRING) =
 DPLIT BYTE (%CHARCOUNT (STRING), %ASCII STRING)%

! \$FAO_TT_OUT constructs a call to FAO with a control string, and some arguments to the control string. This formatted string is then output to the output device.

\$FAO_TT_OUT (CTL_STRING) =
 DBG\$FAO_OUT (\$FAO_STG_COUNT (CTL_STRING)
 %IF %LENGTH GTR 1
 %THEN
 %REMAINING
 %FI
)%;

MISCELLANEOUS LITERALS

This section defines literals which are used throughout the Debugger but which do not belong with any specific data structures.

These four literals are used to index into the DBG\$GB_RADIX byte vector.

LITERAL

DBG\$B_RADIX_INPUT = 0,
DBG\$B_RADIX_OUTPUT = 1,
DBG\$B_RADIX_OUTPUT_OVER = 2;

! Ascii character literals

LITERAL

DBG\$K_LCBIAS	= 32,	! Lower case bias
DBG\$K_TAB	= 9,	! Horizontal tab
DBG\$K_SEMICOLON	= 59,	! ;
DBG\$K_CAR_RETURN	= 13,	! <cr>
DBG\$K_LINE_FEED	= 10,	! <lf>
DBG\$K_NULL	= 0,	! Null character
DBG\$K_EX_POINT	= 33,	! '!'
DBG\$K_BLANK	= 32,	! ' '
DBG\$K_AMPERSAND	= 38,	! '&'
DBG\$K_AT_SIGN	= 64,	! '@'
DBG\$K_SLASH	= 47,	! '/'
DBG\$K_BACKSLASH	= 92,	! '\'
DBG\$K_EQUAL	= 61,	! '='
DBG\$K_COMMA	= 44,	! ','
DBG\$K_DOT	= 46,	! '.'
DBG\$K_COLON	= 58,	! ':'
DBG\$K_UPARROW	= 94,	! '^'
DBG\$K_QUOTE	= 39,	! '''
DBG\$K_LEFT_PARENTHESIS	= 40,	! '('
DBG\$K_RIGHT_PARENTHESIS	= 41,	! ')'
DBG\$K_MULTIPLY	= 42,	! '*'
DBG\$K_PLUS	= 43,	! '+'
DBG\$K_MINUS	= 45,	! '-'
DBG\$K_DIVIDE	= DBG\$K_SLASH,	! '/'
DBG\$K_DBLQUOTE	= 34,	! '"'

! Define unary operators too

DBG\$K_UNARY_PLUS = 230,
DBG\$K_UNARY_MINUS = 231;

! Object type literals. These values are used by the Address Expression Interpreter and the Command Language Interpreter to identity the objects returned by the Address Expression Interpreter. Some of these values also appear in certain descriptors including the Primary, Value, and Address Expression descriptors.

LITERAL

DBG\$K_MIN_DESCR_TYPE = 120, ! Used in FROM clause of CASE

```

0353 0 DBG$K_LITERAL = 120, | Literal value
0354 0 DBG$K_PRIMARY_DESC = 121, | Primary descriptor
0355 0 DBG$K_VALUE_DESC = 122, | Value descriptor
0356 0 DBG$K_PERM_DESC = 123, | Permanent Symbol descriptor
0357 0 DBG$K_INSTRUCTION = 124, | Instruction
0358 0 DBG$K_NC_INSTRUCTION = 125, | Named constant, instruction type
0359 0 DBG$K_NC_OTHER = 126, | Named constant, not instruction
0360 0 DBG$K_OTHER = 127, | Language-specific data type
0361 0 DBG$K_NOTYPE = 128, | Virtual address without any type
0362 0 DBG$K_EXTERNAL_DESC = 129, | External (printable) format
0363 0 DBG$K_VAX_DESC = 130, | VAX standard descriptor
0364 0 DBG$K_V_VALUE_DESC = 131, | Volatile Value descriptor
0365 0 DBG$K_AED = 132, | This code appears in the DBG$B_DHDR_TYPE
0366 0 | field of the new primary descriptor
0367 0 | and also in the corresponding place
0368 0 | in the AED to tell that the descriptor
0369 0 | is an AED and not a new-style descriptor.
0370 0 DBG$K_MAX_DESCR_TYPE = 132; | Used in TO clause of CASE.
0371 0
0372 0 | Define the Radix Literals.
0373 0
0374 0 LITERAL
0375 0 DBG$K_DEFAULT = 1, | Default source language radix
0376 0 DBG$K_BINARY = 2, | Binary radix
0377 0 DBG$K_OCTAL = 8, | Octal radix
0378 0 DBG$K_DECIMAL = 10, | Decimal radix
0379 0 DBG$K_HEX = 16; | Hexadecimal radix
0380 0
0381 0
0382 0 | Define Pseudo-Symbol codes.
0383 0
0384 0 LITERAL
0385 0 DBG$K_CURRENT_LOC = 220, | Current location ""
0386 0 DBG$K_PREDECESSOR = 221, | Previous location ""
0387 0 DBG$K_SUCCESOR = 222, | Next location <CR>
0388 0 DBG$K_LAST_VALUE = 223, | Last value ""
0389 0 DBG$K_INDIRECTION = DBG$K_CURRENT_LOC; | An alternative meaning of ""
0390 0 | is indirection
0391 0
0392 0
0393 0
0394 0 | Token type literals. These are used by the Pathname Parser and scanners.
0395 0
0396 0 LITERAL
0397 0 DBG$K_TOK_NULL = 0, | Null or EOL token
0398 0 DBG$K_TOK_INVALID = 1, | Invalid token
0399 0 DBG$K_TOK_LINE = 2, | '%LINE' token
0400 0 DBG$K_TOK_LABEL = 3, | '%LABEL' token
0401 0 DBG$K_TOK_BS = 4, | '\' token
0402 0 DBG$K_TOK_ID = 5, | ID token
0403 0 DBG$K_TOK_INT = 6, | integer token
0404 0 DBG$K_TOK_DOT = 7, | '.' token
0405 0 DBG$K_TOK_REG = 8, | '%REGISTER' token
0406 0 DBG$K_TOK_QNAME = 9, | '%NAME' token
0407 0 DBG$K_TOK_LOWEST = DBG$K_TOK_NULL;
0408 0 DBG$K_TOK_HIGHEST = DBG$K_TOK_QNAME;
0409 0

```



```

0410 0
0411 00
0412 00
0413 00
0414 00
0415 00
0416 00
0417 00
0418 00
0419 00
0420 00
0421 00
0422 00
0423 00
0424 00
0425 00
0426 00
0427 00
0428 00
0429 00
0430 00
0431 00
0432 00
0433 00
0434 00
0435 00
0436 00
0437 00
0438 00
0439 00
0440 00
0441 00
0442 00
0443 00
0444 00
0445 00
0446 00
0447 00
0448 00
0449 00
0450 00
0451 00
0452 00
0453 00
0454 00
0455 00
0456 00
0457 00
0458 00
0459 00
0460 00
0461 00
0462 00
0463 00
0464 00
0465 00
0466 0

! Return state values for the Pathname Parser
! LITERAL
DBGSK_PN          = 0,      ! Data reference or lexical reference
DBGSK_REG         = 1,      ! Register reference
DBGSK_LINE        = 2,      ! '%LINE' reference
DBGSK_LABEL       = 3,      ! '%LABEL' reference

! Command verb literals. These are used by the Command Line Interpreter.
! LITERAL
DBGSK_ALLOCATE_VERB = 22,    ! ALLOCATE
DBGSK_AT_SIGN_VERB  = 1,     ! @
DBGSK_ATTACH_VERB   = 26,    ! ATTACH
DBGSK_CALL_VERB     = 2,     ! CALL
DBGSK_CANCEL_VERB   = 3,     ! CANCEL
DBGSK_DECLARE_VERB  = 20,    ! DECLARE
DBGSK_DEFINE_VERB   = 4,     ! DEFINE
DBGSK_DELETE_VERB   = 32,    ! DELETE
DBGSK_DEPOSIT_VERB  = 5,     ! DEPOSIT
DBGSK_DISPLAY_VERB  = 28,    ! DISPLAY
DBGSK_DUMP_VERB     = 27,    ! DUMP (Only for developer use)
DBGSK_EDIT_VERB     = 33,    ! EDIT
DBGSK_EVALUATE_VERB = 6,     ! EVALUATE
DBGSK_EXAMINE_VERB  = 7,     ! EXAMINE
DBGSK_EXIT_VERB     = 8,     ! EXIT
DBGSK_EXITLOOP_VERB = 19,    ! EXITLOOP
DBGSK_FOR_VERB      = 25,    ! FOR
DBGSK_GO_VERB       = 9,     ! GO
DBGSK_HELP_VERB     = 13,    ! HELP
DBGSK_IF_VERB       = 16,    ! IF
DBGSK_REPEAT_VERB   = 18,    ! DO
DBGSK_SAVE_VERB     = 31,    ! SAVE
DBGSK_SCROLL_VERB   = 29,    ! SCROLL
DBGSK_SEARCH_VERB   = 15,    ! SEARCH
DBGSK_SELECT_VERB   = 30,    ! SELECT
DBGSK_SET_VERB      = 10,    ! SET
DBGSK_SHOW_VERB     = 11,    ! SHOW
DBGSK_SPAWN_VERB    = 21,    ! SPAWN
DBGSK_STEP_VERB     = 12,    ! STEP
DBGSK_SYMBOLIZE_VERB = 23,    ! SYMBOLIZE
DBGSK_TYPE_VERB     = 14,    ! TYPE
DBGSK_UNDEFINE_VERB = 24,    ! UNDEFINE
DBGSK_WHILE_VERB    = 17,    ! WHILE

DBGSK_FIRST_VERB   = DBGSK_AT_SIGN_VERB,
DBGSK_LAST_VERB    = DBGSK_EDIT_VERB,

EVENTSK_SET_BREAK  = 1,      ! Also SET_BREAK in DBGNSSET
EVENTSK_SET_BREAK_EXC = 3,    ! Also SET_EXCEPTION BREAK in DBGNSSET
EVENTSK_SET_TRACE  = 14,     ! Also SET_TRACE in DBGNSSET
EVENTSK_SET_WATCH  = 17,     ! Also SET_WATCH in DBGNSSET
EVENTSK_SET_STEP   = 11,     ! Also SET_STEP in DBGNSSET
EVENTSK_STEP       = 12,     ! Also DBGSK_STEP_VERB

```

```

0467 0      EVENTSK_SHOW_BREAK      = 1,      ! Also SHOW_BREAK in DBGNSHOW
0468 0      EVENTSK_SHOW_TRACE      = 11,     ! Also SHOW_TRACE in DBGNSHOW
0469 0      EVENTSK_SHOW_WATCH      = 14,     ! Also SHOW_WATCH in DBGNSHOW
0470 0
0471 0      EVENTSK_CANCEL_BREAK     = 2,      ! Also CANCEL_BREAK in DBGNCANCL
0472 0      EVENTSK_CANCEL_BREAK_EXC = 4,      ! Also CANCEL_EXCEPTION_BREAK in DBGNCANCL
0473 0      EVENTSK_CANCEL_TRACE     = 9,      ! Also CANCEL_TRACE in DBGNCANCL
0474 0      EVENTSK_CANCEL_WATCH     = 14;     ! Also CANCEL_WATCH in DBGNCANCL
0475 0
0476 0
0477 0      ! Value kind literals. These values are returned as "value kinds" by routines
0478 0      ! DBG$STA_SYMVALUE and DBG$STA_VALSPEC.
0479 0
0480 0      LITERAL
0481 0          DBG$K_VAL_NOVALUE      = 0,      ! Symbol is a type and has no value
0482 0          DBG$K_VAL_LITERAL      = 1,      ! Value is a literal
0483 0          DBG$K_VAL_ADDR         = 2,      ! Value is an address
0484 0          DBG$K_VAL_DESCR        = 3,      ! Value is a descriptor pointer
0485 0          DBG$K_VAL_UNALLOC      = 4;      ! Symbol was never allocated and
0486 0                                     ! hence has no value
0487 0
0488 0
0489 0      ! Debug specific DTYPE codes which are used in DBGNEXMNE.
0490 0      ! These codes are TEMPORARY!
0491 0
0492 0      LITERAL
0493 0          DBG$K_DTYPE_AD          = 56;     ! ASCID string
0494 0
0495 0
0496 0      ! Maximum number of digits in a packed decimal number.
0497 0
0498 0      LITERAL
0499 0          DBG$K_LARGEST_PACKED    = 31;
0500 0
0501 0
0502 0      ! PL/I specific data type codes. These are referenced in DBGPERMOP and in
0503 0      ! DBGEVALOP in calls to the PL/I RTL during type conversions and calculations.
0504 0
0505 0      LITERAL
0506 0          DBG$K_PLI_PIC            = 1,
0507 0          DBG$K_PLI_FIX_BIN         = 2,
0508 0          DBG$K_PLI_FLO_BIN         = 3,
0509 0          DBG$K_PLI_FIX_DEC         = 4,
0510 0          DBG$K_PLI_FLO_DEC        = 5,
0511 0          DBG$K_PLI_CHAR            = 10,
0512 0          DBG$K_PLI_CHAR_VAR        = 11,
0513 0          DBG$K_PLI_UBIT           = 12,
0514 0          DBG$K_PLI_ABIT           = 14;
0515 0
0516 0
0517 0      ! Debug Print Control Code.
0518 0
0519 0      LITERAL
0520 0          DBG$K_PRTSET_LMARGIN      = 1,      ! Set left margin for DBG$PRINT
0521 0          DBG$K_PRTSET_RLMARGIN     = 2,      ! Set relative left margin
0522 0          DBG$K_PRTBRK_ON_BLANKS    = 3,      ! Set break on blanks flag
0523 0          DBG$K_PRTSET_CONTINUE     = 4,      ! Set indentation length for continuation

```


L 12
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 13
(6)

: 0524 0
: 0525 0 !**-- DBGSK_PRT_RESET = 5; ! Reset to default settings

```
.. 0526 0 ! Input string descriptor definition. This definition is compatible with the
.. 0527 0 ! use of the DSC$B_DTYPE, DSC$B_CLASS, DSC$W_LENGTH, and DSC$A_POINTER macros.
.. 0528 0
.. 0529 0
.. 0530 0 LITERAL
.. 0531 0 DBG$K_STG_DESC_SIZE = 12; ! Length of descriptor in bytes
.. 0532 0
.. 0533 0 MACRO
DBG$STG_DESC = BLOCK [DBG$K_STG_DESC_SIZE, BYTE] %;
```


ADDRESS DESCRIPTORS

The Address Descriptor is used to define an address where a bit offset is needed in addition to a byte address.

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

```

0 +-----+
  |          DBG$L_ADDRESS_BYTE_ADDR          |
  +-----+
1 +-----+
  |          DBG$L_ADDRESS_BIT_OFFSET          |
  +-----+

```

A pointer to an Address Descriptor is declared as follows:

```
ADDPTR: REF DBG$ADDRESS_DESC;
```

Define the fields in the Address Descriptor. Also define the declaration macro.

```

***
FIELD DBG$ADDRESS_DESC_FIELDS =
    SET
    DBG$L_ADDRESS_BYTE_ADDR = [ 0, L_ ], ! Byte address
    DBG$L_ADDRESS_BIT_OFFSET = [ 1, L_ ], ! Bit offset
    TES;

```

```

LITERAL
DBG$K_ADDRESS_DESC_SIZE = 2;          ! Size of descriptor in longwords

```

```

MACRO
DBG$ADDRESS_DESC = BLOCK[DBG$K_ADDRESS_DESC_SIZE]
FIELD(DBG$ADDRESS_DESC_FIELDS) %;

```

!***-

```

: 0534 0
: 0535 0
: 0536 0
: 0537 0
: 0538 0
: 0539 0
: 0540 0
: 0541 0
: 0542 0
: 0543 0
: 0544 0
: 0545 0
: 0546 0
: 0547 0
: 0548 0
: 0549 0
: 0550 0
: 0551 0
: 0552 0
: 0553 0
: 0554 0
: 0555 0
: 0556 0
: 0557 0
: 0558 0
: 0559 0
: 0560 0
: 0561 0
: 0562 0
: 0563 0
: 0564 0
: 0565 0
: 0566 0
: 0567 0
: 0568 0
: 0569 0
: M 0570 0
: 0571 0
: 0572 0

```

DBG\$K_PRIMARY_DESC	The value field contains the address of a primary descriptor
DBG\$K_PERM_DESC	The value field contains the address of a permanent symbol descriptor
DBG\$K_INSTRUCTION	The value field contains a PC value
DBG\$K_NOTYPE	The value field contains an untyped L-value

COMMAND EXECUTION TREES

This section contains the definitions for the nodes that are part of a command execution tree. A command execution tree is built when a debug command is parsed. The tree contains a verb node representing the command (e.g., EXAMINE, DEPOSIT, etc.) as the header node. Linked to the verb node are chains of adverb nodes and noun nodes. The noun node chain hangs off the OBJECT_PTR field and the adverb node chain hangs off the ADVERB_PTR field. The exact structure of the tree depends on the command.

VERB NODES

0	!\$B_VERB_COMPOSITE!\$B_VERB_LITERAL!
1	DBG\$L_VERB_ADVERB_PTR
2	DBG\$L_VERB_OBJECT_PTR

FIELD

DBG\$VERB_NODE_FIELDS =
SET

DBG\$B_VERB_LITERAL	= [0, 0, 8, 0],	! First level verb
DBG\$B_VERB_COMPOSITE	= [0, 8, 8, 0],	! Second or third level verb
DBG\$L_VERB_ADVERB_PTR	= [1, 0, 32, 0],	! Pointer to adverb node
DBG\$L_VERB_OBJECT_PTR	= [2, 0, 32, 0]	! Pointer to object (noun node)

TES;

LITERAL

DBG\$K_VERB_NODE_SIZE = 3; ! Length in longwords

MACRO

DBG\$VERB_NODE = BLOCK [DBG\$K_VERB_NODE_SIZE] FIELD (DBG\$VERB_NODE_FIELDS) %;

ADVERB NODES

0	!ADVERB_LITERAL!
1	DBG\$L_ADVERB_VALUE
2	DBG\$L_ADVERB_LINK

FIELD

DBG\$ADVERB_NODE_FIELDS =
SET

DBG\$B_ADVERB_LITERAL	= [0, 0, 8, 0],	! Adverb id literal
DBG\$L_ADVERB_VALUE	= [1, 0, 32, 0],	! Value (integer)
DBG\$L_ADVERB_LINK	= [2, 0, 32, 0]	! Link to next adverb node

TES;

```

0681 0
0682 0
0683 0
0684 0
0685 0
0686 0
0687 0
0688 0
0689 0
0690 0
0691 0
0692 0
0693 0
0694 0
0695 0
0696 0
0697 0
0698 0
0699 0
0700 0
0701 0
0702 0
0703 0
0704 0
0705 0
0706 0
0707 0
0708 0
0709 0
0710 0
0711 0
0712 0
0713 0
0714 0
0715 0
0716 0
0717 0

```

```

LITERAL  DBG$K_ADVERB_NODE_SIZE = 3;           ! Length in longwords

MACRO     DBG$ADVERB_NODE = BLOCK [DBG$K_ADVERB_NODE_SIZE] FIELD (DBG$ADVERB_NODE_FIELDS) %;

```

```

! NOUN NODES
0  +-----+
   |          DBG$NOUN_VALUE          |
   +-----+
1  +-----+
   |          DBG$ADJECTIVE_PTR       |
   +-----+
2  +-----+
   |          DBG$NOUN_LINK           |
   +-----+
3  +-----+
   |          DBG$NOUN_VALUE2         |
   +-----+

```

```

FIELD DBG$NOUN_NODE_FIELDS =
SET
DBG$NOUN_VALUE      = [ 0, L_ ], ! Noun value
DBG$ADJECTIVE_PTR   = [ 1, L_ ], ! Pointer to adjective (unused)
DBG$NOUN_LINK       = [ 2, L_ ], ! Pointer to next noun
DBG$NOUN_VALUE2     = [ 3, L_ ], ! Additional Noun value
DBG$NOUN_VALUE3     = [ 4, L_ ], ! More Noun value (long nodes only)
DBG$NOUN_VALUE4     = [ 5, L_ ], ! More Noun value (long nodes only)
TES;

```

```

LITERAL  DBG$K_NOUN_NODE_SIZE = 4; ! Length of normal Noun Node in longwords
          DBG$K_NOUN_NODE_SIZE_LONG = 6; ! Length of long Noun Node in longwords

```

```

MACRO     DBG$NOUN_NODE = BLOCK [DBG$K_NOUN_NODE_SIZE] FIELD (DBG$NOUN_NODE_FIELDS) %;

```


COMMAND INPUT STREAM

The Command Input Stream (CIS) is a linked list that the debugger maintains to describe where it is currently taking input from.

The format of a Command Input Stream Descriptor is:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	Unused	SB_INPUT_TYPE	CISW_LENGTH
1	CIS_A_INPUT_PTR		
2	CIS_A_NEXT_LINK		
3	CIS_A_INIT_ADDR		
4	Unused	Flags	CISW_INIT_LENGTH
5	CIS_A_WHILE_CLAUSE		
6	CISL_FOR_UPPER_BOUND	or	CISL_REPEAT_COUNT
7	CIS_A_FOR_LOOP_VAR		
8	CISL_FOR_LOOP_INCR		
9	CISL_SCREEN_OUTPUT		
10	CISL_SCREEN_SOURCE		
11	CISL_SCREEN_ERROR		
12	CIS_A_BUFLIST		
13	CISW_WHILE_LENGTH		

A pointer to a command input stream descriptor block is declared with:

CIS_PTR: REF CISLINK

Define the fields, the descriptor size, and the declaration macro.

FIELD CIS_FIELDS =

```

SET
CISB_INPUT_TYPE = [ 2, B_ ], ! Type of CIS link
CISW_LENGTH     = [ 0, W_ ], ! Length of input buffer
CIS_A_INPUT_PTR = [ 4, L_ ], ! Pointer to input buffer
CIS_A_NEXT_LINK = [ 8, L_ ], ! Ptr to next cis link

```

```

0775 0      CISSA_INIT_ADDR = [ 12, L- ], ! Ptr to start of input buffer
0776 0      CISSW_INIT_LENGTH = [ 16, W- ], ! Allocated buffer size
0777 0      CISSV_REM_FLAG = [ 18, V-(0) ], ! Is link flagged for removal?
0778 0      CISSV_WHILE_FLAG = [ 18, V-(1) ], ! TRUE when condition is true
0779 0      CISSV_SCREEN_NOGO = [ 18, V-(2) ], ! Saved NOGO flag to turn off STEPs
0780 0      ! and GOs in CIS_SCREEN entries
0781 0      CISSA_WHILE_CLAUSE = [ 20, L- ], ! Pointer to WHILE command
0782 0      CISSL_REPEAT_COUNT = [ 24, L- ], ! Count for REPEAT loops
0783 0      CISSL_FOR_UPPER_BOUND = [ 24, L- ], ! Upper bound for FOR loops
0784 0      CISSA_FOR_LOOP_VAR = [ 28, L- ], ! Pointer to the FOR loop var name
0785 0      CISSL_FOR_LOOP_INCR = [ 32, L- ], ! Increment for FOR loops
0786 0      CISSL_SCREEN_OUTPUT = [ 36, L- ], ! Reset value for DBG$GL_SCREEN_OUTPUT
0787 0      CISSL_SCREEN_SOURCE = [ 40, L- ], ! Reset value for DBG$GL_SCREEN_SOURCE
0788 0      CISSL_SCREEN_ERROR = [ 44, L- ], ! Reset value for DBG$GL_SCREEN_ERROR
0789 0      CISSA_BUFLIST = [ 48, L- ], ! List of buffers to free in CIS_REMOVE
0790 0      CISSW_WHILE_LENGTH = [ 52, W- ]
0791 0      TES;

```

```

0792 0
0793 0 LITERAL CIS_ELEMENTS = 54; ! CIS block size (bytes)
0794 0
0795 0

```

```

0796 0 MACRO CIS$LINK = BLOCK [CIS_ELEMENTS, BYTE] FIELD(CIS_FIELDS) %;
0797 0
0798 0
0799 0

```

```

0800 0 ! The following are legal types for CIS links.
0801 0

```

```

0802 0 LITERAL
0803 0     DBG$K_CIS_MINIMUM = 0, ! Lowest type CIS
0804 0     DBG$K_CIS_DBG$INPUT = 0, ! Type DBG$INPUT
0805 0     DBG$K_CIS_RAB = 1, ! Type RAB
0806 0     DBG$K_CIS_INPBUF = 2, ! Type input buffer
0807 0     DBG$K_CIS_ACBUF = 3, ! Type action buffer
0808 0     DBG$K_CIS_REPEAT = 4, ! Type repeat buffer
0809 0     DBG$K_CIS_WHILE = 5, ! Type while buffer
0810 0     DBG$K_CIS_IF = 6, ! Type if buffer
0811 0     DBG$K_CIS_FOR = 7, ! Type for buffer
0812 0     DBG$K_CIS_SCREEN = 8, ! Type for Screen Display
0813 0     DBG$K_CIS_MAXIMUM = 8, ! Highest type CIS
0814 0
0815 0

```

```

0816 0 ! The definitions below should eventually be eliminated when all uses of
0817 0 ! them are converted to the DBG$K_ form.
0818 0

```

```

0819 0 LITERAL
0820 0     CIS_DBG$INPUT = 0, ! Type DBG$INPUT
0821 0     CIS_RAB = 1, ! Type RAB
0822 0     CIS_INPBUF = 2, ! Type input buffer
0823 0     CIS_ACBUF = 3, ! Type action buffer
0824 0     CIS_REPEAT = 4, ! Type repeat buffer
0825 0     CIS_WHILE = 5, ! Type while buffer
0826 0     CIS_IF = 6, ! Type if buffer
0827 0     CIS_FOR = 7, ! Type for buffer
0827 0     CIS_SCREEN = 8, ! Type for Screen Display

```


CONTROL FLAGS

Control flags are global DEBUG status flags used to control the overall flow of DEBUG execution. They represent the major state of DEBUG.

MACRO DBG\$CONTROL_FLAGS = BITVECTOR[];

LITERAL

DBG\$V_CONTROL_TDBG	= 0.	Set if this is a testable DEBUG
DBG\$V_CONTROL_SDBG	= 1.	Set if this is SUPERDEBUG
DBG\$V_CONTROL_KDBG	= 2.	Reserved for future development
DBG\$V_CONTROL_URUN	= 3.	Set if user program has been run to stop DEBUG allocating memory
DBG\$V_CONTROL_EXIT	= 4.	Set if DEBUG is about to EXIT
DBG\$V_CONTROL_FAIL	= 5.	Set by DEBUG internal errors
DBG\$V_CONTROL_DONE	= 6.	Set if user execution complete
DBG\$V_CONTROL_ALLOCATE	= 7.	Set if OK to allocate more memory (e.g., SET MODULE/ALLOCATE)
DBG\$V_CONTROL_USER	= 8.	Set if user program is running
DBG\$V_CONTROL_STOP	= 9.	Set by ^Y,DEBUG sequence to abort processing DEBUG command files
DBG\$V_CONTROL_TBIT	= 10.	Used by DBGEVENT and DBGSTART to control AST/TBIT interactions.
DBG\$V_CONTROL_SCREEN	= 11.	Set if screen displays must be updated because user program has run
DBG\$V_CONTROL_VERSION_4	= 12.	Set if VMS 3B or 4.0 is running

We also define a macro \$ABORT_ON_CONTROL_Y. The purpose of this macro is to allow premature termination of commands which can take a large amount of time to complete but to allow the individual command routines to make sure that they leave all DEBUGs internal data structures in a consistent state. Execution of this macro will have no effect if the Control-Y flag is clear, but will SIGNAL back to DEBUG command level if the flag is set

MACRO \$ABORT_ON_CONTROL_Y =

BEGIN

EXTERNAL DBG\$GV_CONTROL : BITVECTOR[];

IF .DBG\$GV_CONTROL[DBG\$V_CONTROL_STOP] THEN SIGNAL(DBG\$_ABORTED);

END%;

DEFINE SYMBOL TABLE

The Define Symbol Table contains information about the symbols that are currently defined with the DEFINE command.

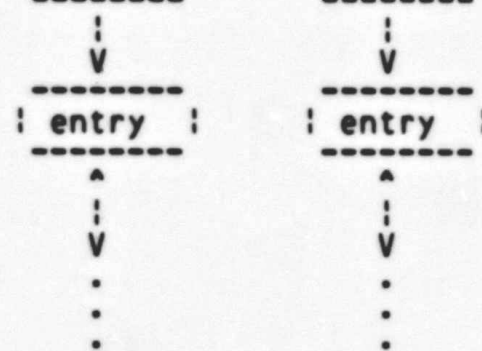
There is a doubly linked list for all globally-defined symbols. (I.e., symbols defined with DEFINE <symbol> == <definition>)
The global variable DBG\$GL_GLOBAL_DEFINE_PTR points to this list.

For each command procedure that is active, there may also be a define list that is local to the procedure. The variable DBG\$GL_LOCAL_DEFINE_PTR points to a doubly-linked list of header blocks. Each header block points to a doubly-linked list of symbols.

Pictorially,

GLOBAL_DEFINE_PTR -> [entry] <-> [entry] <-> ...

LOCAL_DEFINE_PTR -> [header] <-> [header] <-> ...



The format of a header block is:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	+	DEF\$A_NEXT_LINK																			+
1	+	DEF\$A_PREV_LINK																			+
2	+	DEF\$A_DEFINE_LIST																			+

The format of an entry is:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0


```
0927 0
0928 0
0929 0
0930 0
0931 0
0932 0
0933 0
0934 0
0935 0
0936 0
0937 0
0938 0
0939 0
0940 0
0941 0
0942 0
0943 0
0944 0
0945 0
0946 0
0947 0
0948 0
0949 0
0950 0
0951 0
0952 0
0953 0
0954 0
0955 0
0956 0
0957 0
0958 0
0959 0
0960 0
0961 0
0962 0
0963 0
0964 0
0965 0
0966 0
0967 0
0968 0
0969 0
0970 0
0971 0
0972 0
0973 0
0974 0
0975 0
0976 0
0977 0
0978 0
0979 0
0980 0
0981 0
0982 0
0983 0
```

0	DEFSA_ENTRY_NEXT_LINK
1	DEFSA_ENTRY_PREV_LINK
2	DEFSA_NAME
3	DEFSA_VALUE
4	Unused
	DEFSB_KIND

Define the fields of a DEFINE entry.

FIELD DEFINE_ENTRY_FIELDS =

```
SET
DEFSA_ENTRY_NEXT_LINK = [ 0, L_ ],
DEFSA_ENTRY_PREV_LINK = [ 4, L_ ],
DEFSA_NAME             = [ 8, L_ ],
DEFSA_VALUE            = [ 12, L_ ],
DEFSB_KIND             = [ 16, B0_ ]
```

```
Forward link
Backward link
Points to a counted string
with the name being defined
Points to a descriptor or
counted string with the
"value" of the symbol.
Contains a code for the "kind"
of definition. The codes
are defined below.
```

TES;

Define the fields of a header block.

FIELD DEFINE_HEADER_FIELDS =

```
SET
DEFSA_NEXT_LINK        = [ 0, L_ ],
DEFSA_PREV_LINK        = [ 4, L_ ],
DEFSA_DEFINE_LIST      = [ 8, L_ ]
```

```
Forward link
Backward link
Points to the linked list
of DEFINE entries.
```

TES;

Define the size in bytes and in words of the above structures.

LITERAL

```
DBGSK_DEFINE_ENTRY_SIZE_B = 17,
DBGSK_DEFINE_ENTRY_SIZE_W = 5,
DBGSK_DEFINE_HEADER_SIZE_B = 12,
DBGSK_DEFINE_HEADER_SIZE_W = 3;
```

Define the codes that will be used for the possible kinds of symbol definitions.

LITERAL

J 13
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 24
(13)

```
0984 0      DEFINE_LOWEST      = 1,      ! Lowest value of DEFINE codes
0985 0      DEFINE_ADDRESS     = 1,
0986 0      DEFINE_COMMAND     = 2,
0987 0      DEFINE_PROCEDURE   = 3,
0988 0      DEFINE_STRING      = 4,
0989 0      DEFINE_VALUE       = 5,
0990 0      DEFINE_PARAMETER   = 6,      ! This is used to temporarily store strings
0991 0                                     ! representing actual parameters, until
0992 0                                     ! they are bound to formals by the
0993 0                                     ! DECLARE command.
0994 0      DEFINE_KEY          = 7,
0995 0      DEFINE_HIGHEST     = 7;      ! Highest value of DEFINE codes
0996 0
0997 0
0998 0      ! Macros for declaration of DEFINE entries and headers.
0999 0      !
1000 0      MACRO
M 1001 0      DEFINE$ENTRY =
1002 0          BLOCK [DBG$K_DEFINE_ENTRY_SIZE_B, BYTE] FIELD (DEFINE_ENTRY_FIELDS)%,
M 1003 0      DEFINE$HEADER =
1004 0          BLOCK [DBG$K_DEFINE_HEADER_SIZE_B, BYTE] FIELD (DEFINE_HEADER_FIELDS)%;
```


ERROR STATUS CODES

Debugger status codes are defined in the MDL file DBGME.SMDL--new ones should therefore be entered in that file. All that is found here are definitions relating to components of status codes.

Define the possible values of the status severity field.

LITERAL SYS\$K_INFO = 3; ! Informational status severity

1005 0
1006 0
1007 0
1008 0
1009 0
1010 0
1011 0
1012 0
1013 0
1014 0
1015 0
1016 0
1017 0

```

0 :      [5]  EVENTSV_SILENT
1 :      [5]  EVENTSV_THREADED
2 :      [5]  EVENTSV_ONCE_ONLY
3 :      [5]  EVENTSV_HAS_VAL_DSCR
4 :      [5]  EVENTSV_ROUTINE
5 :      [5]  EVENTSV_RET_AT_PC

7 :      [5]  EVENTSV_DELETED

0 :      [6]  EVENTSV_OVRD_LINE
1 :      [6]  EVENTSV_STEP_LINE
2 :      [6]  EVENTSV_OVRD_OVER
3 :      [6]  EVENTSV_STEP_OVER
4 :      [6]  EVENTSV_OVRD_SOURCE

```


M 13
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 27
(15)

1075 0
1076 0
1077 0
1078 0
1079 0
1080 0
1081 0
1082 0
1083 0
1084 0
1085 0
1086 0
1087 0
1088 0
1089 0
1090 0
1091 0
1092 0
1093 0
1094 0
1095 0
1096 0
1097 0
1098 0
1099 0
1100 0
1101 0
1102 0
1103 0
1104 0
1105 0
1106 0
1107 0
1108 0
1109 0
1110 0
1111 0
1112 0
1113 0
1114 0
1115 0
1116 0
1117 0
1118 0
1119 0
1120 0
1121 0
1122 0
1123 0
1124 0
1125 0
1126 0
1127 0
1128 0
1129 0
1130 0
1131 0

5 : [6] EVENTSV_STEP_SOURCE
6 : [6] EVENTSV_OVRD_NOSYSTEM
7 : [6] EVENTSV_STEP_NOSYSTEM
8 : [6] EVENTSV_OVRD_NOSILENT
9 : [6] EVENTSV_STEP_NOSILENT

A : [6] EVENTSV_STEP_BPT
B : [6] EVENTSV_STEP_TICK
C : [6] EVENTSV_STEP_LINE_OK

D : [6] EVENTSV_BREAK_ADDRESS

A pointer to an Event Table Entry is declared as follows:

EVENTPTR: REF EVENTSEVENT_DESCRIPTOR

Define the fields of the Event Table Entry. Also declare the maximum size of an Event Table Entry and define the declaration macro.

FIELD EVENTSEVENT_DESCRIPTOR_FIELDS =

SET
EVENTSA_EVENT_DESCRIPTOR = [0,A_], ! Event Descriptor address

EVENTSL_CMD_FLINK = [0,L_], ! Command queue forward link
EVENTSL_CMD_BLINK = [1,L_], ! Command queue backward link

EVENTSL_EXC_FLINK = [2,L_], ! Exception queue forward link
EVENTSL_EXC_BLINK = [3,L_], ! Exception queue backward link

EVENTSL_EXCEPTION = [4,L_], ! Exception code we expect to see

EVENTSB_CMD_TYPE = [5,B0_], ! Command type
EVENTSB_CMD_KIND = [5,B1_], ! Command kind
EVENTSB_SUB_KIND = [5,B2_], ! Command sub-kind

EVENTSB_CMD_FLAGS = [5,B3_], ! Command flags

EVENTSV_SILENT = [5,V3_(0)], ! Flag set if /SILENT qualifier set
EVENTSV_THREADED = [5,V3_(1)], ! Flag set if this is threaded code
EVENTSV_ONCE_ONLY = [5,V3_(2)], ! Flag set for temporary event entry
EVENTSV_HAS_VAL_DSCR = [5,V3_(3)], ! Flag set if there is an Event Value Descriptor
EVENTSV_ROUTINE = [5,V3_(4)], ! Flag set if ROUTINE EventPoint
EVENTSV_RET_AT_PC = [5,V3_(5)], ! Flag set if SET BREAK/RET (or TRACE) address is at current PC
EVENTSV_DELETED = [5,V3_(7)], ! Flag set if this entry is deleted

EVENTSL_STEP_FLAGS = [6,L_], ! 'Stepping' flags

EVENTSV_OVRD_LINE = [6,V_(0)], ! Flag set if override step/line
EVENTSV_STEP_LINE = [6,V_(1)], ! Flag set if step/line
EVENTSV_OVRD_OVER = [6,V_(2)], ! Flag set if override step/over
EVENTSV_STEP_OVER = [6,V_(3)], ! Flag set if step/over
EVENTSV_OVRD_SOURCE = [6,V_(4)], ! Flag set if override step/source
EVENTSV_STEP_SOURCE = [6,V_(5)], ! Flag set if step/source
EVENTSV_OVRD_NOSYSTEM = [6,V_(6)], ! Flag set if override step/nosystem


```

1132 0 EVENTSV_STEP_NOSYSTEM = [6,V_( 7)], ! Flag set if step/nosystem
1133 0 EVENTSV_OVRD_NOSILENT = [6,V_( 8)], ! Flag set if override step/nosilent
1134 0 EVENTSV_STEP_NOSILENT = [6,V_( 9)], ! Flag set if step/nosilent
1135 0
1136 0 EVENTSV_STEP_BPT = [6,V_(10)], ! Flag set if step over BPT active
1137 0 EVENTSV_STEP_TICK = [6,V_(11)], ! Flag set if new step
1138 0 EVENTSV_STEP_NOLINE = [6,V_(12)], ! Flag set if line not found
1139 0
1140 0 EVENTSV_BREAK_ADDRESS = [6,V_(13)], ! Flag set if the prolog address is
1141 0 ! used
1142 0 EVENTSB_SAVED_SUB_KIND = [6,B3_], ! Saved value of SUB_KIND
1143 0
1144 0 EVENTSL_AFTER_COUNT = [7,L_], ! After count
1145 0 EVENTSL_STEP_COUNT = [7,L_], ! Step count
1146 0 EVENTSL_SSV_COUNT = [7,L_], ! Address of a count which keeps
1147 0 ! track of the number of SSV calls
1148 0 ! originated from one SSV
1149 0
1150 0 EVENTSL_WHEN = [8,L_], ! WHEN expression pointer
1151 0 EVENTSL_USERS_PC = [8,L_], ! User's PC for STEP/RETURN
1152 0
1153 0 EVENTSL_DO = [9,L_], ! DO command list pointer
1154 0 EVENTSL_USERS_FP = [9,L_], ! User's FP for STEP/RETURN and
1155 0 ! [BREAK:TRACE]/RETURN skips
1156 0
1157 0 EVENTSL_PRIMARY = [10,L_], ! Primary pointer
1158 0 EVENTSL_OPCODE_LIST = [10,L_], ! Instruction opcode list
1159 0
1160 0 EVENTSL_ADDRESS = [11,L_], ! Byte address
1161 0 EVENTSL_FP_RET_CNT = [11,L_], ! Address of a count which keeps
1162 0 ! track of RET for SSV return levels
1163 0
1164 0 EVENTSL_VALDESC = [12,L_], ! Value descriptor pointer
1165 0
1166 0 EVENTSB_OPCODE = [12,B0_], ! Opcode byte
1167 0
1168 0 EVENTSL_THREAD = [12,L_], ! Thread
1169 0
1170 0 EVENTSA_VMSDESC = [13,A_], ! VMS Descriptor
1171 0
1172 0 EVENTSL_STEP_LO_PC = [13,L_], ! Step line lo PC
1173 0 EVENTSL_STEP_HI_PC = [14,L_], ! Step line hi PC
1174 0
1175 0 EVENTSL_CALL_FRAME = [15,L_] ! Pointer to Saved value from FP
1176 0
1177 0 TES;
1178 0
1179 0 LITERAL
1180 0 EVENT$K_EVENT_DESCRIPTOR_SIZE = 16; ! Descriptor size in longwords
1181 0
1182 0 MACRO
1183 0 EVENT$EVENT_DESCRIPTOR =
1184 0 BLOCK [EVENT$K_EVENT_DESCRIPTOR_SIZE]
1185 0 FIELD (EVENT$EVENT_DESCRIPTOR_FIELDS) %;
1186 0
1187 0
1188 0 ! Macro to assign an increasing value to each member of a <name> list,

```



```

1189 0 ! starting with <constant>.
1190 0
1191 0 MACRO
1192 0     ENUMERATE (CONSTANT, NAME) [] =
1193 0         NAME = CONSTANT
1194 0         %IF %NULL (%REMAINING)
1195 0             %THEN %EXITMACRO
1196 0             %FI
1197 0             , ENUMERATE (CONSTANT + 1, %REMAINING) %;
1198 0
1199 0
1200 0 ! The following are the allowed values of the EVENT$B_CMD_TYPE field.
1201 0
1202 0 LITERAL
1203 0     ENUMERATE(0, ! Starting with 0...
1204 0         EVENT$K_TYPE_BREAK, ! Break
1205 0         EVENT$K_TYPE_TRACE, ! Trace
1206 0         EVENT$K_TYPE_WATCH, ! Watch
1207 0         EVENT$K_TYPE_STEPS, ! Steps
1208 0         EVENT$K_TYPE_SKIPS, ! Skips
1209 0         EVENT$K_TYPE_IGNORE, ! Ignored activating an event
1210 0         EVENT$K_TYPE_SSINT ! System service Intercept Event
1211 0     );
1212 0
1213 0
1214 0 ! The following are the allowed values of the EVENT$B_CMD_KIND field.
1215 0
1216 0 LITERAL
1217 0     ENUMERATE(0, ! Starting with 0...
1218 0         EVENT$K_KIND_ACC, ! Access (/READ, etc.)
1219 0         EVENT$K_KIND_INS, ! Instruction
1220 0         EVENT$K_KIND_EXC, ! Exception
1221 0         EVENT$K_KIND_SKP ! Skips
1222 0     );
1223 0
1224 0
1225 0 ! The following are the allowed values of the EVENT$B_SUB_KIND field.
1226 0
1227 0 LITERAL
1228 0     ENUMERATE(0, ! Starting with 0...
1229 0         EVENT$K_ACC_READ, ! Access Read
1230 0         EVENT$K_ACC_WRIT, ! Access Write
1231 0         EVENT$K_ACC_MDFY, ! Access Modify
1232 0         EVENT$K_ACC_EXEC, ! Access Execute
1233 0         EVENT$K_ACC_RTRN, ! Access Return
1234 0         EVENT$K_INS_CALL, ! Instruction Call
1235 0         EVENT$K_INS_BRAN, ! Instruction Branch
1236 0         EVENT$K_INS_LINE, ! Instruction Line
1237 0         EVENT$K_INS_EVR, ! Instruction Every
1238 0         EVENT$K_INS_USER, ! Instruction User
1239 0         EVENT$K_SKP_READ, ! Skipping Read
1240 0         EVENT$K_SKP_WRIT, ! Skipping Write
1241 0         EVENT$K_SKP_MDFY, ! Skipping Modify
1242 0         EVENT$K_SKP_EXEC, ! Skipping Execute
1243 0         EVENT$K_SKP_RTRN, ! Skipping Return
1244 0         EVENT$K_EXC_EXC
1245 0     );

```


EVENT TABLE ENTRY FOR [BREAK|TRACE] [/RETURN] POINTS

This form of the Event Table Entry is used for ordinary and /RETURN breakpoints and tracepoints. It is thus used whenever the implementation of the eventpoint is to store a BPT instruction at the event location. This form is not used for threaded code (used only for COBOL-74), however.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	EVENT\$L_CMD_FLINK
1	EVENT\$L_CMD_BLINK
2	EVENT\$L_EXC_FLINK
3	EVENT\$L_EXC_BLINK
4	EVENT\$L_EXCEPTION
5	7! - 13!2!1!0! B_SUB_KIND B_CMD_KIND B_CMD_TYPE
6	B_SAVED_SUB_KIND! - unused - !D!C!B!A!9!8!7!6!5!4!3!2!1!0!
7	EVENT\$L_AFTER_COUNT
8	EVENT\$L_WHEN
9	EVENT\$L_DO
10	EVENT\$L_PRIMARY
11	EVENT\$L_ADDRESS
12	Unused EVENT\$B_OPCODE
13	EVENT\$A_VMSDESC
14	
15	

EVENT TABLE ENTRY FOR /READ, /WRITE, /MODIFY AND WATCHPOINTS

This form of the Event Table Entry is used in all cases when data locations are watched for read accesses, write accesses, or modifying accesses. It is thus used both for SET BREAK/MODIFY, for example, and for the SET WATCH command.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	EVENT\$L_CMD_FLINK
1	EVENT\$L_CMD_BLINK
2	EVENT\$L_EXC_FLINK
3	EVENT\$L_EXC_BLINK
4	EVENT\$L_EXCEPTION
5	7! - 13!2!1!0! B_SUB_KIND B_CMD_KIND B_CMD_TYPE
6	B_SAVED_SUB_KIND! - unused - !C!B!A!9!8!7!6!5!4!3!2!1!0!
7	EVENT\$L_AFTER_COUNT
8	EVENT\$L_WHEN
9	EVENT\$L_DO
10	EVENT\$L_PRIMARY
11	EVENT\$L_ADDRESS
12	EVENT\$L_VALDESC
13	EVENT\$A_VMSDESC
14	
15	

EVENT TABLE ENTRY FOR /CALL, /BRANCH, /INSTRUCTION

This form of Event Table Entry is used for the SET BREAK and SET TRACE commands when the /CALL, /BRANCH, or /INSTRUCTION qualifier is present. It is thus used in all cases where instructions must be traced (T-bitted), one instruction at a time, and the event is activated when an instruction of a certain class is encountered.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	EVENT\$L_CMD_FLINK
1	EVENT\$L_CMD_BLINK
2	EVENT\$L_EXC_FLINK
3	EVENT\$L_EXC_BLINK
4	EVENT\$L_EXCEPTION
5	7: - 13:2:1:0: B_SUB_KIND : B_CMD_KIND : B_CMD_TYPE
6	B_SAVED_SUB_KIND: - unused - :C:B:A:9:8:7:6:5:4:3:2:1:0:
7	EVENT\$L_AFTER_COUNT
8	EVENT\$L_WHEN
9	EVENT\$L_DO
10	EVENT\$L_OPCODE_LIST
11	EVENT\$L_ADDRESS
12	Unused :EVENT\$B_OPCODE
13	EVENT\$L_STEP_LO_PC
14	EVENT\$L_STEP_HI_PC

EVENT TABLE ENTRY FOR STEP

This form of Event Table Entry is used for the STEP command.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	EVENT\$L_CMD_FLINK
1	EVENT\$L_CMD_BLINK
2	EVENT\$L_EXC_FLINK
3	EVENT\$L_EXC_BLINK
4	EVENT\$L_EXCEPTION
5	7! - !3!2!1!0! B_SUB_KIND ! B_CMD_KIND ! B_CMD_TYPE
6	!B_SAVED_SUB_KIND! - unused - !C!B!A!9!8!7!6!5!4!3!2!1!0!
7	EVENT\$L_STEP_COUNT
8	EVENT\$L_USERS_PC
9	EVENT\$L_USERS_FP
10	EVENT\$L_OPCODE_LIST
11	EVENT\$L_ADDRESS
12	Unused !EVENT\$B_OPCODE
13	EVENT\$L_STEP_LO_PC
14	EVENT\$L_STEP_HI_PC
15	EVENT\$L_CALL_FRAME

EVENT STEPPING DESCRIPTOR

This structure defines the current step type(s).

```

      3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
      1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0  +-----+-----+-----+-----+-----+-----+
   |XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX|^!%$#!a!|*|
   +-----+-----+-----+-----+-----+-----+
1  |                                     EVENT$SL_STEPPING_OP_LIST
   +-----+-----+-----+-----+-----+-----+

```

```

* : [0] EVENT$B_STEPPING_KIND
@ : [0] EVENT$V_STEPPING_LINE
# : [0] EVENT$V_STEPPING_OVER
$ : [0] EVENT$V_STEPPING_SOURCE
% : [0] EVENT$V_STEPPING_NOSYSTEM
^ : [0] EVENT$V_STEPPING_NOSILENT

```

A pointer to an Step Descriptor Entry is declared as follows:

```
STEPPINGPTR: REF EVENT$STEPPING_DESCRIPTOR
```

Define the fields of the Step Descriptor Entry. Also declare the maximum size of an Step Descriptor Entry and define the declaration macro.

```

FIELD EVENT$STEPPING_DESC_FIELDS =
SET
EVENT$A_STEPPING_DESCRIPTOR = [0,A] | Event Descriptor address
EVENT$B_STEPPING_KIND = [0,B0] | Step kind (as EVENT$B_SUB_KIND)
EVENT$V_STEPPING_LINE = [0,V-8] | Stepping /LINE
EVENT$V_STEPPING_OVER = [0,V-9] | Stepping /OVER
EVENT$V_STEPPING_SOURCE = [0,V-10] | Stepping /SOURCE
EVENT$V_STEPPING_NOSYSTEM = [0,V-11] | Stepping /NOSYSTEM
EVENT$V_STEPPING_NOSILENT = [0,V-12] | Stepping /NOSILENT
EVENT$SL_STEPPING_OP_LIST = [1,L_] | /CALL, /BRANCH, etc.
TES;

```

```

LITERAL
EVENT$K_STEPPING_DESC_SIZE = 2; | Descriptor size in longwords

```

```

MACRO
EVENT$STEPPING_DESCRIPTOR =
BLOCK [EVENT$K_STEPPING_DESC_SIZE]
FIELD (EVENT$STEPPING_DESC_FIELDS) %;

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0

0	!	EVENT\$\$_DO_LIST_FLINK
1	!	EVENT\$\$_DO_LIST_BLINK
2	!	EVENT\$\$_DO_LIST_COUNT
3	!	EVENT\$\$_DO_LIST_POINT

A pointer to an Event DO List Descriptor is declared as follows:

DOPTR: REF EVENT\$DO_LIST_DESCRIPTOR

Define the fields of the Event DO List Descriptor. Also declare the descriptor size and define the declaration macro.

```

FIELD      EVENT$DO_LIST_DESCRIPTOR_FIELDS =
SET
EVENT$A_DO_LIST_DESCRIPTOR = [0,A_], ! DO List Descriptor address

EVENT$L_DO_LIST_FLINK      =   [0,L_], ! DO queue forward
EVENT$L_DO_LIST_BLINK      =   [1,L_], ! and backward links

EVENT$L_DO_LIST_COUNT      =   [2,L_], ! DO reference count

EVENT$L_DO_LIST_POINT      =   [3,L_] ! DO list pointer
TES;

```

```
LITERAL
EVENTSK_DO_LIST_DESCRIPTOR_SIZE = 4; ! Descriptor size (longwords)
```

```
MACRO
EVENTSDO_LIST_DESCRIPTOR =
    BLOCK [EVENTSK_DO_LIST_DESCRIPTOR_SIZE]
    FIELD (EVENTSDO_LIST_DESCRIPTOR_FIELDS) %;
```


An Event WHEN Descriptor is created each time a SET BREAK, SET TRACE, or SET WATCH statement has a WHEN clause. The Event WHEN Descriptor points to the location of the boolean WHEN-expression and allows this expression to be evaluated each time the X-point is reached. This is the format of the Event WHEN Descriptor:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0

0	EVENTSL_WHEN_FLINK
1	EVENTSL_WHEN_BLINK
2	EVENTSL_WHEN_COUNT
3	EVENTSL_WHEN_POINT

WHENPTR: REF EVENTSWHEN_DESCRIPTOR

Define the fields of the Event WHEN Descriptor, its size, and its declaration macro.

FIELD EVENTS WHEN_DESCRIPTOR_FIELDS =

```

EVENTS$WHEN_DESCRIPTOR = [0,A_], ! WHEN Descriptor address
EVENTS$WHEN_FLINK      = [0,L_], ! WHEN queue forward link
EVENTS$WHEN_BLINK      = [1,L_], ! WHEN queue backward link
EVENTS$WHEN_COUNT       = [2,L_], ! WHEN reference count
EVENTS$WHEN_POINT       = [3,L_], ! WHEN list pointer
TES;

```

```

LITERAL
EVENT$K_WHEN_DESCRIPTOR_SIZE =      4; ! Descriptor size (longwords)

```

```
MACRO
EVENTSWHEN_DESCRIPTOR =
    BLOCK [EVENTSK_WHEN_DESCRIPTOR_SIZE]
    FIELD (EVENTSWHEN_DESCRIPTOR_FIELDS) %;
```

An Event Page Descriptor is created for each page which is write-protected to watch-point one or more variables on that page. Event Page Descriptors are linked together on a doubly linked list which is pointed to by variable XXXXXX. This is the format of an Event Page Descriptor:

0	EVENT\$L_PAGE_FLINK
1	EVENT\$L_PAGE_BLINK
2	EVENT\$L_PAGE_ADDRESS
3	Unused PAGE_PROTECTION EVENT\$W_PAGE_REF_COUNT

PAGEDESC: REF EVENT\$PAGE_DESCRIPTOR

FIELD EVENT\$PAGE_DESCRIPTOR_FIELDS =

```

EVENT$PAGE_DESCRIPTOR_FIELDS =
SET
EVENT$A_PAGE_DESCRIPTOR = [0,A_], ! Page Descriptor address
EVENT$L_PAGE_FLINK       = [0,L_], ! Page queue forward link
EVENT$L_PAGE_BLINK       = [1,L_], ! Page queue backward link
EVENT$L_PAGE_ADDRESS      = [2,L_], ! Page byte address
EVENT$W_PAGE_REF_COUNT    = [3,W0], ! Page referecne count
EVENT$B_PAGE_PROTECTION   = [3,B2], ! Page old protection
TES:

```

```

LITERAL
EVENT$K_PAGE_DESCRIPTOR_SIZE =      4; ! Descriptor size (longwords)

```

```
MACRO
EVENT$PAGE_DESCRIPTOR =
    BLOCK [EVENT$K_PAGE_DESCRIPTOR_SIZE]
    FIELD (EVENT$PAGE_DESCRIPTOR_FIELDS) %;
```


FREE MEMORY MANAGEMENT

The Free Memory Manager manages the Debugger's free memory pool. Its routines are thus called throughout the Debugger to allocate and de-allocate memory blocks used for RST entries, Static Address Table entries, and numerous other kinds of descriptors and records. The Free Memory Manager has two kinds of blocks in its memory pool: free blocks and allocated blocks. The formats of both kinds of blocks and the over-all structure of the memory pool is illustrated below.

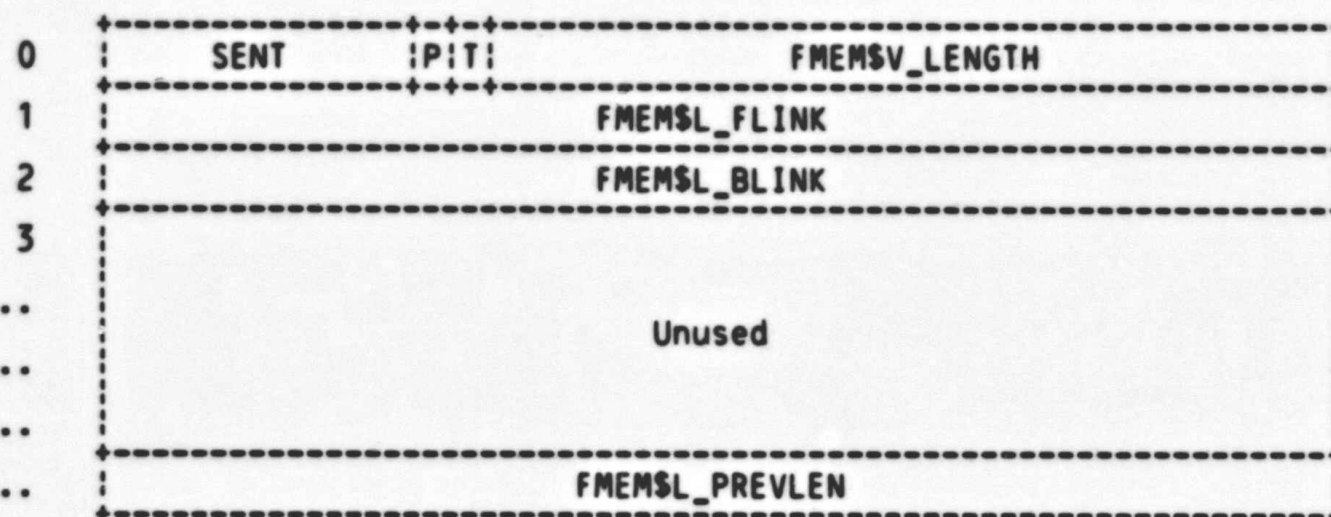
FORMAT OF A FREE MEMORY BLOCK

The format of a free memory block is shown here:

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```



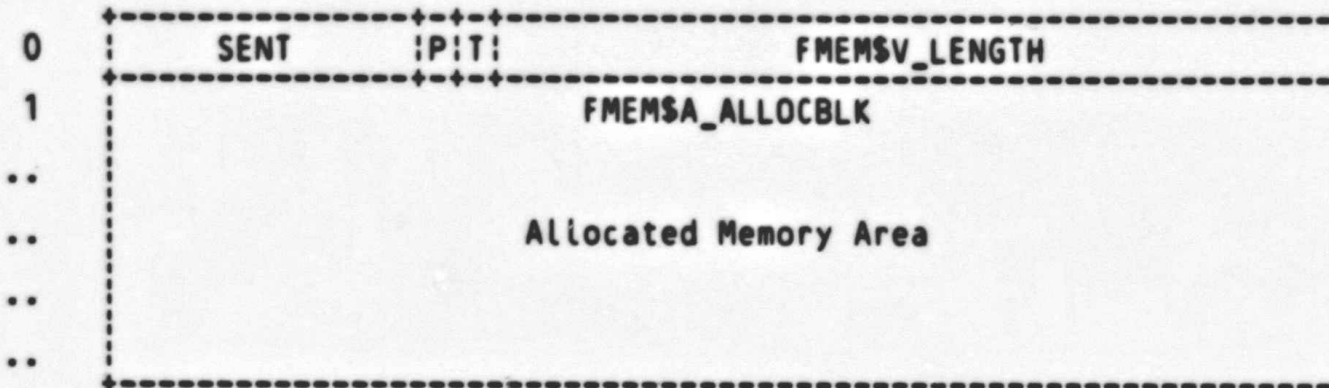
Here SENT represents FMEMSB_SENTINEL, P is FMEMSV_PREVALLOC, and T is FMEMSV_THISALLOC as defined on the next page.

A free block has its length (in longwords) stored both at the beginning of the block (FMEMSV_LENGTH) and at the end (FMEMSL_PREVLEN). The minimum length of a free block is four longwords. Free blocks are chained together on a doubly linked free-list. This list has a list head which is never allocated--the list is thus never empty. The address of the free-list list head is always stored in the OWN variable DBG\$FREE_LIST.

FORMAT OF AN ALLOCATED MEMORY BLOCK

The format of an allocated block is as follows:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



Here SENT represents FMEMSB_SENTINEL, P is FMEMSV_PREVALLOC, and T is FMEMSV_THISALLOC as defined below. In an allocated block, the address returned to the requestor is represented by FMEMSA_ALLOCBLK.

A pointer to a memory block (free or allocated) is declared as follows:

BLKPTR: REF FMEMSBLOCK

This declaration is used in the Free Memory Manager only.

Define the fields in the two kinds of memory blocks.

FIELD FMEM\$FLD_DEF =

SET		
FMEMSV_LENGTH	= [0, V_(0,22)]	! The length of the block in longwords
FMEMSV_THISALLOC	= [0, V_(22)]	! Set to TRUE if this block is allocated
FMEMSV_PREVALLOC	= [0, V_(23)]	! Set to TRUE if the previous block is allocated (block at smaller addr)
FMEMSB_SENTINEL	= [0, B3_]	Sentinel value--always FMEM\$K_SENTINEL
FMEMSL_FLINK	= [1, L_]	Forward link on free list
FMEMSL_BLINK	= [2, L_]	Backward link on free list
FMEMSL_PREVLEN	= [-1, L_]	Length of this free block relative to the start of the next block
FMEMSA_ALLOCBLK	= [1, A_]	Address returned to caller requesting a new memory block
FMEMSA_HEADER	= [-1, A_]	Address of block's header relative to the FMEMSA_ALLOCBLK location

TES;

N 14
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 41
(26)

```
1785 0
1786 0
1787 0
1788 0
1789 0
1790 0
1791 0
1792 0
1793 0

! Declare the sentinel value and the declaration macro.
LITERAL
    FMEMSK_SENTINEL = %X'B2';      ! Magic sentinel value--used for error
                                   ! checking only
MACRO
    FMEMSBLOCK = BLOCK[,LONG] FIELD(FMEM$FLD_DEF) %;
```


OVERALL STRUCTURE OF MEMORY POOL

The memory pool consists of one or more memory pool areas. The first such area is initialized to have the format shown here:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	Forward link to next memory pool area
1	Size in bytes of this memory pool area
2	SENT 1101 4
3	FMEMSL_FLINK
4	FMEMSL_BLINK
5	4
6	SENT 1101 FMEMSV_LENGTH
7	FMEMSL_FLINK
8	FMEMSL_BLINK
9	
..	
..	
..	Free memory available for allocation
..	
..	
..	
..	FMEMSL_PREVLEN
..	SENT 1011 1

Here longword 0 contains a pointer to the next memory pool area (for this initial area the pointer is always zero) and longword 1 contains the size of this area in bytes. These two fields are present for internal error checking purposes only.

Longwords 2 - 5 contain the free-list list head. This is a free block which is always at the start of the doubly linked free list and which is never coalesced with any other free block--the code thus never has


```

: 1851 0
: 1852 00
: 1853 00
: 1854 00
: 1855 00
: 1856 00
: 1857 00
: 1858 00
: 1859 00
: 1860 00
: 1861 00
: 1862 00
: 1863 00
: 1864 00
: 1865 00
: 1866 00
: 1867 00
: 1868 00
: 1869 00
: 1870 00
: 1871 00
: 1872 00
: 1873 00
: 1874 00
: 1875 00
: 1876 00
: 1877 00
: 1878 00
: 1879 00
: 1880 00
: 1881 00
: 1882 00
: 1883 00
: 1884 00
: 1885 00
: 1886 00
: 1887 00
: 1888 00
: 1889 00
: 1890 00
: 1891 00
: 1892 00
: 1893 00
: 1894 00
: 1895 0

```

to worry about a completely empty free list. The very last longword in the memory pool area is the header of an allocated block. It prevents any attempt to coalesce free blocks off the end of the area. The rest of the area (longword 6 to the next to last longword) is initially one big free block from which allocation can take place. Note how the P and T flags (FMEMSV_PREVALLOC and FMEMSV_THISALLOC) are set to prevent free block coalescing off either end of the memory pool area.

Additional memory pool areas may be acquired from LIB\$GET_VM during the Debugger's initialization phase. Each such area is initialized to have the same format as the first one except that the free-list list head is absent--only one is needed for the whole memory pool. The initial format of each additional memory pool area is illustrated here:

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

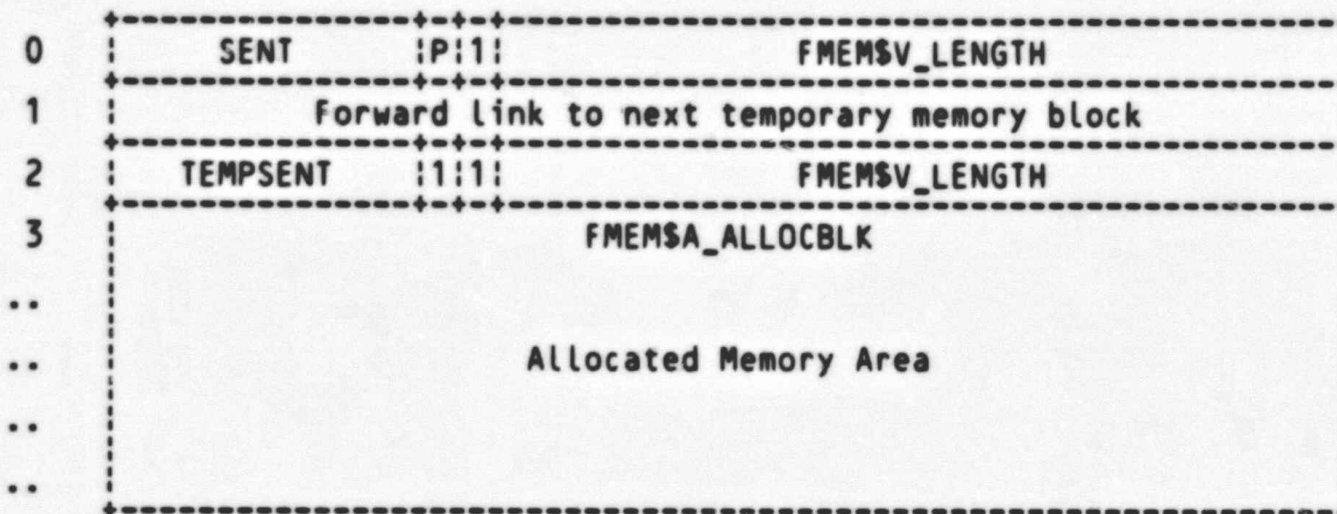
```

0	Forward link to next memory pool area
1	Size in bytes of this memory pool area
2	SENT !1!0! FMEMSV_LENGTH
3	FMEMSL_FLINK
4	FMEMSL_BLINK
5	
..	
..	
..	Free memory available for allocation
..	
..	
..	FMEMSL_PREVLEN
..	SENT !0!1! 1

TEMPORARY MEMORY BLOCKS

"Temporary" memory blocks are blocks which automatically are released at the end of the current command. Such blocks are allocated by a special allocation routine (DBG\$GET_TEMPMEM), but they do not have to be explicitly released to the memory pool. Instead one routine is called (DBG\$REL_TEMPMEM) which automatically releases all "temporary" blocks accumulated since the last such call. Temporary blocks have the following format:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



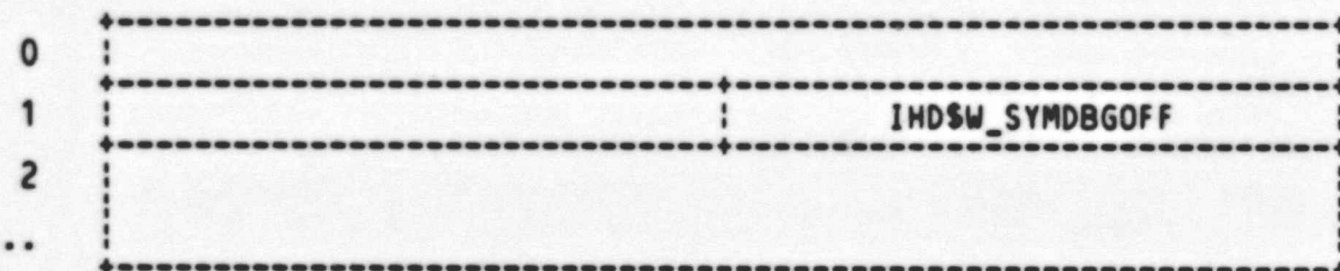
Longword 0 is the same as for any other memory pool block. Longword 1 contains a forward link to the next block on the temporary block list. Longword 2 is similar to longword 0, but the length there is two less and the sentinel value is a different sentinel value (FMEM\$K_TEMPSENT).

Define the distinguishing sentinel value for "temporary" memory blocks.

LITERAL FMEM\$K_TEMPSENT = %X'B4'; ! Sentinel value in longword 2

The executable image file header block contains a pointer in a fixed location which points to a small block later in the header which gives the size and location of the Debug Symbol Table (DST) and the Global Symbol Table (GST). The first part of the executable image file header looks as follows:

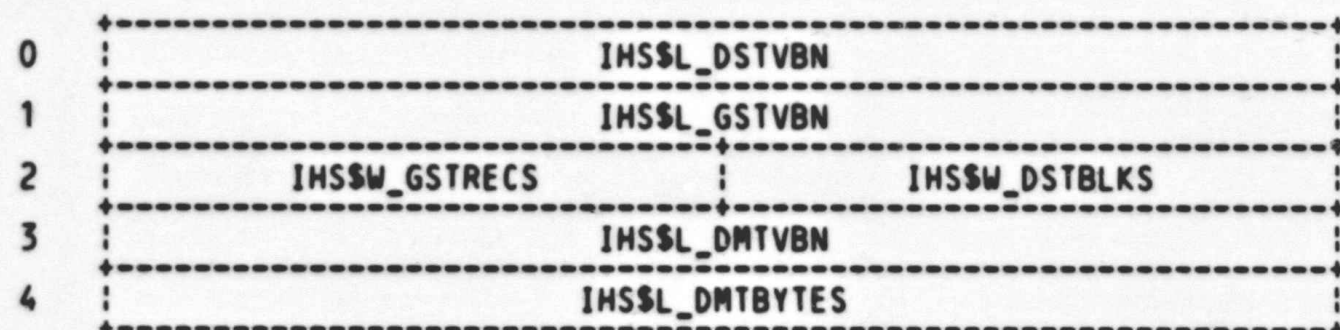
3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



Here IHDSW_SYMDBGOFF contains the byte offset relative to the start of the header of an Image Header Symbol Table descriptor.

The Image Header Symbol Table Descriptor (IHS) pointed to by the IHDSW_SYMDBGOFF field in the header has the following format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



Here IHSSW_DSTBLKS and IHSSL_DSTVBN give the size (in blocks) and location (Virtual Block Number) of the Debug Symbol Table (DST) within the


```

1999 0
2000 0
2001 0
2002 0
2003 0
2004 0
2005 0
2006 0
2007 0
2008 0
2009 0
2010 0
2011 0
2012 0
2013 0
2014 0
2015 0
2016 0
2017 0
2018 0
2019 0
2020 0
2021 0

```

executable image file. The fields IHSSW GSTRECS and IHSSL GSTVBN give the size (in GST records) and start location (Virtual Block Number) of the Global Symbol Table (GST). Finally, the fields IHSSL DMTBYTES and IHSSL DMTVBN give the size (in bytes) and start location (Virtual Block Number) of the Debug Module Table (DMT). The DMT is described below. These field names are declared by macros in SYSSLIBRARY:LIB.L32.

The symbol IHDSW SYMDBGOFF is defined in SYSSLIBRARY:LIB.L32 and is therefore not defined here. A pointer to the image file header is declared as follows:

```
IHDPTR: REF BLOCK[,BYTE]
```

A pointer to the image header symbol descriptor is declared like this:

```
IHSPTR: REF IHSENTRY
```

Define the IHSENTRY macro.

MACRO

```
IHSENTRY = BLOCK[IHSSK_LENGTH,BYTE] %; ! IHSSK_LENGTH is defined in
! SYSSLIBRARY:LIB.L32
```


IMAGE FILE DEBUG MODULE TABLE

The Image File Header in an executable image file points to the Image Header Symbol Table descriptor as described above. If bit 5 of field IHDSL_LNKFLAGS in the image header is set, this is a "new" image, i.e. one produced by the VMS V4.0 or later linker, and the IHSSL_DMTVBN and IHSSL_DMTBYTES fields exist in the Image Header Symbol Table descriptor. (If bit 5 is not set, this is an "old" image and those fields do not exist.) If non-zero, IHSSL_DMTVBN gives the Virtual Block Number in the image file of the Debug Module Table (the DMT). IHSSL_DMTBYTES then gives the size of the DMT in bytes. The DMT is only built if the user did a LINK/DEBUG; if he did not, IHSSL_DMTVBN and IHSSL_DMTBYTES are zero.

The Debug Module Table contains one entry per module in the Debug Symbol Table (the DST). This is the format of each such DMT entry:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DST address of Module Begin DST Record
1	Size in bytes of module's DST
2	Number of PSECTs for this module
3	Start address of first PSECT in module
4	Length of first PSECT in module in bytes
..	
..	(Two longwords per PSECT)
..	
..	Start address of last PSECT in module
..	Length of last PSECT in module in bytes

Longword 0 gives the address relative to the start of the DST of the Module Begin DST Record for this module. Longword 1 gives the size of the DST in bytes for the same module. Longword 2 gives the number of PSECTs in the module (i.e., the number of statically allocated program sections), and this is followed by that number of two-longword pairs which give the start address and length (in bytes) of each such PSECT. Since the number of PSECTs cannot exceed 65K, the upper two bytes of longword 2 are available for future expansion.

The DMT is used during DEBUG initialization to initialize the RST and

: 2079 0
: 2080 0
: 2081 0
: 2082 0
: 2083 0
: 2084 0
: 2085 0
: 2086 0

the Program Static Address Table. Using the DMT is much faster than the alternative procedure, namely reading through the entire DST to pick up the needed information. The information in the DMT entry is enough to build a Module RST Entry for each module in the DST and the PSECT information is used to build the Program SAT. The amount of RST symbol table space needed per module is not computable from the DMP information, but is estimated by multiplying the DST size of each module by an appropriate scale factor.

LEAST RECENTLY USED MODULE TABLE

The Least Recently Used Module (LRUM) table keeps track of the Least Recently Used module in the RST. This information is used to select modules to be removed from the RST when the free memory pool is full and new space must be created by releasing old RST entries. The LRUM table is built and maintained by the DBG\$MOST RECENT MODULE routine in module RSTACCESS and is accessed by routine DBG\$GET MEMORY when memory space must be released by removing the Least Recently Used module. Routine DBG\$RST_INIT builds the initial LRUM list head.

The LRUM table consists of a doubly linked list of LRUM entries, with a permanent list head and one entry per module currently in the RST. The address of the list head is given by the global LRUM\$LISTHEAD. The format of the individual LRUM entry is show here:

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	+	-----	+
	:	LRUM\$L_FLINK	:
1	+	-----	+
	:	LRUM\$B_LINK	:
2	+	-----	+
	:	LRUM\$RSTPTR	:
	+	-----	+

A pointer to a Least Recently Used Module entry is declared as follows:

```
LRUMPTR: REF LRUM$ENTRY
```

Define the fields in the Least Recently Used Module entry. Also define the entry size and the declaration macro.

```

FIELD LRUM$FLD_DEF =
  SET
    LRUM$FLINK = [ 0, L_ ], ! The LRUM chain forward link
    LRUM$B_LINK = [ 1, L_ ], ! The LRUM chain backward link
    LRUM$RSTPTR = [ 2, L_ ], ! A pointer to the Module RST Entry
  TES;

LITERAL
  LRUM$ENTSIZE = 3; ! The longword size of the LRUM entry

MACRO
  LRUM$ENTRY = BLOCK[,LONG] FIELD(LRUM$FLD_DEF) %; ! Declaration macro

```


LEXICAL SCANNER CHARACTER TABLE

The Lexical Scanner (routine DBGSLEXICAL_SCANNER) operates on a Character Table which gives lexical information about each ASCII character. This information consists of bits and small integer values which characterize the character from the Lexical Scanner's point of view--they indicate whether a given character can be part of an identifier, a numeric constant, an operator, or whatever. The Character Table is thus a vector of 256 elements, indexed by the ASCII character code, where each element contains the information defined here.

A Lexical Scanner Character Table is declared as follows:

CHARTBL: CHRTBL\$TABLE

Define the fields of the Character Table element. Also define the declaration macro.

FIELD CHRTBL\$FLD_DEF =

SET		
CHRTBL\$V_IDENT_START	= [0, V_(0)],	Identifier start character
CHRTBL\$V_IDENT_MIDDLE	= [0, V_(1)],	Identifier middle character
CHRTBL\$V_IDENT_END	= [0, V_(2)],	Identifier end character
CHRTBL\$V_NUMBER_START	= [0, V_(3)],	Number start character
CHRTBL\$V_NUMBER_CLASS	= [0, V_(4,4)],	Numeric constant character class
CHRTBL\$V_ALPHABETIC	= [0, V_(8)],	Alphabetic character (A-Z)
CHRTBL\$V_DIGIT	= [0, V_(9)],	Numeric digit (0-9)
CHRTBL\$V_STRING_QUOTE	= [0, V_(10)],	String quote character
CHRTBL\$V_OPCHAR	= [0, V_(11)],	Operator character
CHRTBL\$V_OPCHAR_PREFIX	= [0, V_(12)],	Operator prefix character
CHRTBL\$V_OPCHAR infix	= [0, V_(13)],	Operator infix character
CHRTBL\$V_OPCHAR_POSTFIX	= [0, V_(14)],	Operator postfix character
CHRTBL\$V_ADDRESS_OP	= [0, V_(15)],	Address Expression operator
CHRTBL\$V_SPECIAL_SYMBOL	= [0, V_(16)],	Special DEBUG symbol (. \ ^)
CHRTBL\$V_SPACE	= [0, V_(17)],	Space or Tab character
CHRTBL\$V_TERMINATOR	= [0, V_(18)],	May start Terminator Token
CHRTBL\$V_SPECIAL_CASE	= [0, V_(19)],	Special case in lex scanner
CHRTBL\$L_WHOLE_ENTRY	= [0, L_]	The whole entry for a char.

TES;

MACRO

CHRTBL\$TABLE = BLOCKVECTOR[256,1, LONG] FIELD(CHRTBL\$FLD_DEF) %;

! Define mask values for all the bits in the Character Table element. These values are used to initialize Character Table elements.

LITERAL

CHRTBL\$M_NOTHING	= 0,	! No bits set
CHRTBL\$M_IDENT_START	= 1<0,	! Identifier start character
CHRTBL\$M_IDENT_MIDDLE	= 1<1,	! Identifier middle character

K 15
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 51
(32)

```
2197 0 CHRTBL$M_IDENT_END = 1^2, : Identifier end character
2198 00 CHRTBL$M_IDENT_ANYWHERE = 7^0, : Anywhere in an identifier
2199 00 CHRTBL$M_NUMBER_START = 1^3, : Number start character
2200 00 CHRTBL$M_ALPHABETIC = 1^8, : Alphabetic character (A-Z)
2201 00 CHRTBL$M_DIGIT = 1^9, : Numeric digit (0-9)
2202 00 CHRTBL$M_STRING_QUOTE = 1^10, : String quote character
2203 00 CHRTBL$M_OPCHAR = 1^11, : Operator character
2204 000 CHRTBL$M_OPCHAR_PREFIX = 1^12, : Operator prefix character
2205 000 CHRTBL$M_OPCHAR infix = 1^13, : Operator infix character
2206 000 CHRTBL$M_OPCHAR_POSTFIX = 1^14, : Operator postfix character
2207 00 CHRTBL$M_ADDRESS_OP = 1^15, : Address Expression operator
2208 00 CHRTBL$M_SPECIAL_SYMBOL = 1^16, : Special DEBUG symbol (. \ ^)
2209 00 CHRTBL$M_SPACE = 1^17, : Space or tab character
2210 000 CHRTBL$M_TERMINATOR = 1^18, : May start terminator token
2211 000 CHRTBL$M_SPECIAL_CASE = 1^19, : Special case in lexical scanner
2212 000
2213 000
2214 00 : Note that the possible values of the CHRTBL$V NUMBER CLASS field are defined
2215 00 : under the description of the Number Scanner State Table. They are the same
2216 00 : values as are allowed in the NUMST$B_CHAR_CLASS field and their names all
2217 0 : have the form NUMST$K_CLASS_xxx.
```


TOKEN\$B_KIND	= [0, B0_],	Token kind
TOKEN\$B_FLAGS	= [0, B1_],	Token Entry flag bits
TOKEN\$V_PRIMARY	= [0, V1_(0)],	Token is Primary Symbol operator (used in Operator entries only)
TOKEN\$V_LEXICAL	= [0, V1_(1)],	Token is a lexical operator (used in Operator entries only)
TOKEN\$V_SGNEXT	= [0, V1_(2)],	Sign extension S in X<P,S,E> operation (used in Operator Lexical Token for the bit-select operation)
TOKEN\$V_BALANCED_PARENS	= [0, V1_(0)],	Token is Terminator Token only if parentheses are balanced (used in Terminator entries only)
TOKEN\$V_MUST_BE_SINGLE	= [0, V1_(1)],	Token is Terminator Token only if this character is not followed by it- self (: is term but not ::) (used in Terminator entries only)
TOKEN\$V_ARGUMENT_LIST	= [0, V1_(3)],	Argument list follows this Ada tick operator.
TOKEN\$W_CODE	= [0, W1_],	Operand or operator code for token
TOKEN\$B_BIF	= [1, B0_],	Built-in Function Code
TOKEN\$B_LENGTH	= [2, B0_],	Length of operand name in bytes

M 15
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 53
(33)

```
2275 0      TOKEN$A_NAME      = [ 2, A1_ ],      ! Operand name in ASCII
2276 0
2277 0      TOKEN$B_R_PREC     = [ 1, B0_ ],      ! Operator's Right Precedence
2278 0      TOKEN$B_L_PREC     = [ 1, B1_ ],      ! Operator's Left Precedence
2279 0      TOKEN$W_SUBCODE    = [ 1, W1_ ],      ! Sub-code field
2280 0      TOKEN$W_BIT_OFFSET = [ 2, W0_ ],      ! Offset P in X<P,S,E> operation
2281 0
2282 0      TOKEN$W_BIT_LENGTH  = [ 2, W1_ ],      ! Length S in X<P,S,E> operation
2283 0
2284 0      TOKEN$B_OPLEN      = [ 3, B0_ ],      ! Length of operator name in bytes
2285 0      TOKEN$A_OPNAME     = [ 3, A1_ ],      ! Operator name in ASCII
2286 0      TES;
2287 0
2288 0      LITERAL
2289 0      TOKEN$K_ENTSIZE = 2,      ! Size of fixed part of an Operand
2290 0      ! Lexical Token Entry
2291 0      TOKEN$K_ENTSIZE_OPERATOR = 3;      ! Size of fixed part of an Operator
2292 0      ! Lexical Token Entry
2293 0
2294 0      MACRO
2295 0      TOKEN$ENTRY = BLOCK[,LONG] FIELD(TOKEN$FLD_DEF) %;      ! Declaration macro
2296 0
2297 0      ! Define the valid values of the TOKEN$B_KIND field. This field indicates
2298 0      ! what kind of token this is.
2299 0
2300 0      LITERAL
2301 0      TOKEN$K_OPERAND      = 1,      ! Operand
2302 0      TOKEN$K_PREFIX_OP    = 2,      ! Prefix Operator
2303 0      TOKEN$K infix_OP     = 3,      ! Infix Operator
2304 0      TOKEN$K_POSTFIX_OP   = 4;      ! Postfix Operator
2305 0
2306 0
2307 0      ! Define mask values for the bits in the TOKEN$B_FLAGS field.
2308 0
2309 0      LITERAL
2310 0      TOKEN$M_PRIMARY      = 1^0,      ! Token is Primary Symbol operator
2311 0      TOKEN$M_LEXICAL      = 1^1,      ! Token is lexical operator
2312 0      TOKEN$M_BALANCED_PARENS = 1^0,      ! Terminator requires balanced parens
2313 0      TOKEN$M_MUST_BE_SINGLE = 1^1;      ! Terminator character must be single
```


Operands are identifiers, numeric constants, string or character constants, or special constructs such as %LINE nnn which generate identifiers. The Lexical Token Entry for an operand has the following format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	TOKEN\$W_CODE	TOKEN\$B_FLAGS	TOKEN\$B_KIND
1	Unused		TOKEN\$B_BIF
2	TOKEN\$A_NAME		TOKEN\$B_LENGTH
..	(Operand name in Counted ASCII)		
..	(Includes TOKEN\$B_LENGTH and TOKEN\$A_NAME fields)		
..			

The TOKENSB KIND field of an Operand Lexical Token Entry always has the value TOKENSK_OPERAND.

Define the valid values of the TOKEN\$W_CODE field for operands. This field says what kind of operand this Token Entry represents.

```

LITERAL
TOKENSK_MIN_OPERAND      = 1.      | --- Minimum value for TOKENSW_CODE
TOKENSK_IDENTIFIER      = 1.      | Identifier
TOKENSK_STRING           = 2.      | Character String Constant
TOKENSK_CHARACTER        = 3.      | Single Character Constant
TOKENSK_INTEGER          = 4.      | Integer Constant
TOKENSK_HEX_INTEGER      = 5.      | Hexadecimal Integer Constant
TOKENSK_FLOATING         = 6.      | Floating Point Constant
TOKENSK_EXP_E_FLOAT      = 7.      | Floating Constant with E Exponent
TOKENSK_EXP_D_FLOAT      = 8.      | Floating Constant with D Exponent
TOKENSK_EXP_Q_FLOAT      = 9.      | Floating Constant with Q Exponent
TOKENSK_BIN_INTEGER      = 10.     | Binary Constant
TOKENSK_OCT_INTEGER      = 11.     | Octal Constant
TOKENSK_COMPLEMENT       = 12.     | 1's Complement
TOKENSK_BIT_STRING       = 13.     | Bit String Constant (as in PL/I)
TOKENSK_PACK_DECIMAL     = 14.     | Packed Decimal Constant
TOKENSK_EXP_G_FLOAT      = 15.     | Floating Constant with G Exponent
TOKENSK_BUILTIN_FUNCTION = 16.     | Built-in Function
TOKENSK_MAX_OPERAND      = 16:     | --- Maximum value for TOKENSW_CODE

```


OPERATOR LEXICAL TOKEN ENTRY

The Lexical Token Entry for an operator contains additional information not needed for operands, namely the left and right precedences of the operator. This is the format of an Operator Lexical Token Entry:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1

1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	TOKEN\$W_CODE	TOKEN\$B_FLAGS	TOKEN\$B_KIND
1	TOKEN\$W_SUBCODE	TOKEN\$B_L_PREC	TOKEN\$B_R_PREC
2	TOKEN\$W_BIT_LENGTH	TOKEN\$W_BIT_OFFSET	
3	TOKEN\$A_OPNAME	TOKEN\$B_OPLEN	
..	(Operator name string in Counted ASCII)		
..	(Includes TOKEN\$B_OPLEN and TOKEN\$A_OPNAME fields)		
..			

The TOKEN\$B_KIND field of an Operator Lexical Token Entry always has one of the three values TOKEN\$K_PREFIX_OP, TOKEN\$K infix_OP, or TOKEN\$K postfix_OP.

Define some size literals used for allocating memory.

LITERAL

TOKEN\$K_FIXED_SIZE_BYTE = 13,

TOKEN\$K_FIXED_SIZE_LONG = 4;

Fixed size (not including the name) of an operator token. Actual size in bytes is this value plus the OPLEN field.

Fixed size (not including the name) of an operator token. Actual size in longwords is this value plus OPLEN / 4.

Define the valid values of the TOKEN\$W_CODE field for operators. This field indicates which operation is to be performed. Not all languages allow all operators, but all operators allowed in at least one language are listed in this table. Note that the same operator may be represented differently in different languages--thus "=" and ".EQ." are both considered to be the same comparison operator.


```

2427 0 !
2428 0 LITERAL
2429 0
2430 0 TOKEN$K_MIN_PRIMARY_OP = 1, --- Minimum value of TOKEN$W_CODE
2431 0 --- for Primary Symbol operators
2432 0 TOKEN$K_PRIMARY_TERM = 1, Terminator operator for primary
2433 0 TOKEN$K_GLOBAL_SLASH = 2, Prefix backslash for GST scope (\X)
2434 0 TOKEN$K_BACKSLASH = 3, Infix backslash for pathname qualifi-
2435 0 --- cation (EXAM M\R\X)
2436 0 TOKEN$K_SUBSCRIPT = 4, Subscript open parenthesis operator
2437 0 TOKEN$K_DOT = 5, Infix data qualification operator
2438 0 --- as in A.B(2).C in PL/I
2439 0 TOKEN$K_COBOL_OF = 6, Infix data qualification operator
2440 0 --- as in C OF B OF A(2) in COBOL
2441 0 TOKEN$K_PASCAL_DEREF = 7, Postfix dereference operator such as
2442 0 --- ^ in PASCAL (as in P^.X)
2443 0 TOKEN$K_PLI_DEREF = 8, Infix dereference operator such as
2444 0 --- -> in PL/I (as in P->X)
2445 0 TOKEN$K_INVOCNUM = 9, Invocation Number operator
2446 0 TOKEN$K_BIF_OP = 10, Built-in Function (parameter list)
2447 0 TOKEN$K_ADA_TICK = 11, The ADA tick operator for symbol
2448 0 --- attribute qualification.
2449 0 TOKEN$K_MAX_PRIMARY_OP = 11; --- Maximum value for TOKEN$W_CODE
2450 0 --- for Primary Symbol operators
2451 0
2452 0 LITERAL
2453 0 TOKEN$K_MIN_OPERATOR = 1, --- Minimum value for TOKEN$W_CODE
2454 0 --- for language expression operators
2455 0 TOKEN$K_INITIATOR = 1, Expression initiator pseudo-operator
2456 0 TOKEN$K_TERMINATOR = 2, Expression terminator pseudo-operator
2457 0 TOKEN$K_INDIRECT = 3, Indirection operator (prefix . or @)
2458 0 TOKEN$K_UNARY_PLUS = 4, Unary plus operator (+A)
2459 0 TOKEN$K_UNARY_MINUS = 5, Unary minus operator (-A)
2460 0 TOKEN$K_ADD = 6, Addition operator (A+B)
2461 0 TOKEN$K_SUBTRACT = 7, Subtraction operator (A-B)
2462 0 TOKEN$K_MULTIPLY = 8, Multiplication operator (A*B)
2463 0 TOKEN$K_DIVIDE = 9, Division operator (A/B)
2464 0 TOKEN$K_POWER_OF = 10, Exponentiation operator (A**B)
2465 0 TOKEN$K_OPENPAREN = 11, Open expression parenthesis "("
2466 0 TOKEN$K_CLOSEPAREN = 12, Close expression parenthesis ")"
2467 0 TOKEN$K_EQUAL = 13, Equal comparison operator
2468 0 TOKEN$K_NOT_EQUAL = 14, Not equal comparison operator
2469 0 TOKEN$K_GTR_THAN = 15, Greater than comparison operator
2470 0 TOKEN$K_GTR_THAN_U = 16, Unsigned Greater than comparison operator
2471 0 TOKEN$K_GTR_EQUAL = 17, Greater than or equal comparison oper.
2472 0 TOKEN$K_GTR_EQUAL_U = 18, Unsigned Greater than or equal to comparison operator
2473 0 TOKEN$K_LSS_THAN = 19, Less than comparison operator
2474 0 TOKEN$K_LSS_THAN_U = 20, Unsigned Less than comparison operator
2475 0 TOKEN$K_LSS_EQUAL = 21, Less than or equal comparison oper.
2476 0 TOKEN$K_LSS_EQUAL_U = 22, Unsigned Less than or equal comparison operator
2477 0 TOKEN$K_NOT = 23, Boolean NOT operator
2478 0 TOKEN$K_AND = 24, Boolean AND operator
2479 0 TOKEN$K_OR = 25, Boolean OR operator
2480 0 TOKEN$K_XOR = 26, Boolean XOR (exclusive OR) operator
2481 0 TOKEN$K_EQV = 27, Boolean EQV (equivalence) operator
2482 0 TOKEN$K_SHORT_AND = 28, Short-circuited boolean AND operator
2483 0 TOKEN$K_SHORT_OR = 29, Short-circuited boolean OR operator

```


2484	0	TOKENSK_BIT_NOT	= 30,	Bitwise NOT operator
2485	0	TOKENSK_BIT_AND	= 31,	Bitwise AND operator
2486	0	TOKENSK_BIT_OR	= 32,	Bitwise OR operator
2487	0	TOKENSK_BIT_XOR	= 33,	Bitwise XOR operator
2488	0	TOKENSK_BIT_EQV	= 34,	Bitwise EQV operator
2489	0	TOKENSK_CONCATENATE	= 35,	String concatenation
2490	0	TOKENSK_MODULUS	= 36,	Modulus operator
2491	0	TOKENSK_REMAINDER	= 37,	Remainder operator
2492	0	TOKENSK_LEFT_SHIFT	= 38,	Left shift operator
2493	0	TOKENSK_RIGHT_SHIFT	= 39,	Right shift operator
2494	0	TOKENSK_ADDRESS_OF	= 40,	Address-of operator (as in C)
2495	0	TOKENSK_SIZEOF	= 41,	Size-of operator (as in C)
2496	0	TOKENSK_SET_MEMBER	= 42,	Set membership operator
2497	0	TOKENSK_ABSOLUTE	= 43,	Absolute value operator
2498	0	TOKENSK_INFIX_NOT	= 44,	Lexical operator (as in COBOL's NOT =)
2499	0	TOKENSK_SUCCESOR	= 45,	Successor operator (BIF)
2500	0	TOKENSK_PREDECESSOR	= 46,	Predecessor operator (BIF)
2501	0	! Unused	= 47,	Unused
2502	0	TOKENSK_INT_DIVIDE	= 48,	Lexical operator for integer divide (in PASCAL, A DIV B must be distinguished from A / B)
2503	0			
2504	0	TOKENSK_BITSELECT	= 49,	Bit selection in BLISS and address expressions (<pos,size,ext>)
2505	0			
2506	0	TOKENSK_DEPOSIT	= 50,	= for Deposit Command (acting as lang. dependent assignment)
2507	0			
2508	0	TOKENSK_CONVERT	= 51,	Used as to indicate subscript conversion operator
2509	0			
2510	0	TOKENSK_RADIX_DEC	= 52,	Set current radix to decimal
2511	0	TOKENSK_RADIX_HEX	= 53,	Set current radix to hexadecimal
2512	0	TOKENSK_RADIX_OCT	= 54,	Set current radix to octal
2513	0	TOKENSK_RADIX_BIN	= 55,	Set current radix to binary
2514	0	TOKENSK_IDENTITY	= 56,	Identity operator
2515	0	TOKENSK_OPENSET	= 57,	Set constant open [
2516	0	TOKENSK_BIT_IMP	= 58,	Bitwise IMP (implication) operator
2517	0	TOKENSK_PRE_INCR	= 59,	Pre-increment (++X)
2518	0	TOKENSK_POST_INCR	= 60,	Post-increment (X++)
2519	0	TOKENSK_PRE_DECR	= 61,	Pre-decrement (--X)
2520	0	TOKENSK_POST_DECR	= 62,	Post-decrement (X--)
2521	0	TOKENSK_PREFIX_GTR	= 63,	Prefix > as in COBOL, ie. NOT >
2522	0	TOKENSK_PREFIX_LSS	= 64,	Prefix < as in COBOL, ie. NOT <
2523	0	TOKENSK_PREFIX_EQL	= 65,	Prefix = as in COBOL, ie. NOT =
2524	0	TOKENSK_NEGCONST	= 66,	Negative constant
2525	0	TOKENSK_POSCONST	= 67,	Positive constant
2526	0	TOKENSK_MAX_OPERATOR	= 67;	---
2527	0			Maximum value of TOKENSW_CODE
2528	0			---
2529	0			for language expression operators

Define the possible values for the TOKENSK_SUBCODE field.
This field is used to store additional information about the operator.
At present the one use of this field is for the ADA attribute operators.
For these, the TOKENSW_CODE field contains "TOKENSK_ADA_TICK" and the
subcode field identifies which tick operator is present ("FIRST",
"LAST", etc.
The reason for doing it this way is because there may be a large number
of tick operators, all of which are treated the same for purposes of
operator precedence parsing. Tables thus are smaller if there is only
one code for "tick" and a subcode for each tick operator.

E 16
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 58
(35)

```
: 2541 0 LITERAL
: 2542 00
: 2543 00 TOKEN$K_TICK_MIN = 1,
: 2544 00
: 2545 00 TOKEN$K_TICK_CONSTRAINED= 1,
: 2546 00 TOKEN$K_TICK_FIRST = 2,
: 2547 00 TOKEN$K_TICK_LAST = 3,
: 2548 00 TOKEN$K_TICK_LENGTH = 4,
: 2549 00 TOKEN$K_TICK_POS = 5,
: 2550 00 TOKEN$K_TICK_PRED = 6,
: 2551 00 TOKEN$K_TICK_SIZE = 7,
: 2552 00 TOKEN$K_TICK_SUCC = 8,
: 2553 00 TOKEN$K_TICK_VAL = 9,
: 2554 0 TOKEN$K_TICK_MAX = 9;
```


TERMINATOR LEXICAL TOKEN ENTRY

Terminators are lexical tokens which mark the end of an expression. For example, in subscript expressions, "(", ")", and "}" mark the ends of the expressions as in "ARR(K+2, I+3-4)". Similarly, in the command SET BREAK X+4 DO(...), the keyword "DO" marks the end of the expression as long as it does not appear within parentheses. Terminator Lexical Token Entries define tokens which act as expression terminators in specified contexts. The Terminator Lexical Token Entry has the following format:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	TOKEN\$W_CODE	TOKEN\$B_FLAGS	TOKEN\$B_KIND
1	unused		
2	TOKEN\$A_NAME	TOKEN\$B_LENGTH	
..	(Terminator name in Counted ASCII)		
..	(Includes TOKEN\$B_LENGTH and TOKEN\$A_NAME fields)		
..			

The TOKEN\$B_KIND field of a Terminator Lexical Token Entry always has the value zero (0).

The TOKEN\$B_FLAGS field can contain the flag TOKEN\$V_BALANCED_PARENS which, if set, indicates that this token acts as a terminator only if parentheses are balanced when it is encountered. For example, "}" terminates a subscript expression only the subscript expression's parentheses are balanced so far. Similarly, DO terminates the address expression in SET BREAK expr DO ... only if it is not enclosed in parentheses; otherwise there would be no way to include the identifier "DO" (as allowed in many languages) in the address expression.

Define the valid values of the TOKEN\$W_CODE field for terminators. This field says what kind of terminator this Token Entry represents.

LITERAL

TOKEN\$K_MIN_TERMINATOR	=	0.	---	Minimum value for TOKEN\$W_CODE
TOKEN\$K_TERM_NONE	=	0.		No terminator (just end of line)
TOKEN\$K_TERM_COMMA	=	1.		Comma ","
TOKEN\$K_TERM_CLOSE	=	2.		Close paren or bracket (")" or "]"
TOKEN\$K_TERM_COLON	=	3.		Colon ":"
TOKEN\$K_TERM_EQUAL	=	4.		Equal sign "="

:	2612	0	TOKEN\$K_TERM_DO	=	5,	DO keyword "DO" (as in SET BREAK)
:	2613	0	TOKEN\$K_TERM_THEN	=	6,	THEN keyword "THEN" (as in IF - THEN)
:	2614	0	TOKEN\$K_TERM_WHEN	=	7,	WHEN keyword "WHEN" (as in SET BREAK)
:	2615	0	TOKEN\$K_TERM_COMCOL	=	8,	Allow both comma "," and colon ":"
:	2616	0	TOKEN\$K_TERM_CMWDO	=	9,	Allow comma, WHEN, and DO (SET BREAK)
:	2617	0	TOKEN\$K_TERM_GTRTHAN	=	10,	Greater-than sign ">" (as in <p,s,e>)
:	2618	0	TOKEN\$K_TERM_OPEN	=	11,	Open paren "("
:	2619	0	TOKEN\$K_TERM_COMPAREN	=	12,	"." or ")" in CALL
:	2620	0	TOKEN\$K_TERM_TO	=	13,	"TO" in FOR i=1 TO n
:	2621	0	TOKEN\$K_TERM_BY	=	14,	"BY"
:	2622	0	TOKEN\$K_TERM_DOT	=	15,	Dot "." (as in Set constant, ie. 1..4)
:	2623	0	TOKEN\$K_MAX_TERMINATOR	=	15;	--- Maximum value for TOKEN\$W_CODE

LINKED LIST NODE

Linked lists using this definition are generated by language specific routines to indicate:

1. The pages on which a symbol's R-value is contained.
(DBG\$NXXX_GET_PAGES)
2. Primary Descriptors which represent a range specification.
(DBG\$NXXX_RANGE_VAL)

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	-----	DBG\$LINK_NODE_LINK	-----
1	-----	DBG\$LINK_NODE_VALUE	-----

A pointer to a Link Node is declared as follows:

LINKNODE: REF DBG\$LINK_NODE

Define the Link Node fields and declaration macro.

FIELD DBG\$LINK_NODE_FIELDS =

SET
DBG\$LINK_NODE_LINK = [0, L_], ! Pointer to next node
DBG\$LINK_NODE_VALUE = [1, L_], ! Node value field
TES;

LITERAL

DBG\$K_LINK_NODE_SIZE = 2; ! Size of link node in
! longwords

MACRO

DBG\$LINK_NODE = BLOCK [DBG\$K_LINK_NODE_SIZE]
FIELD (DBG\$LINK_NODE_FIELDS) %;

!***

2624 0
2625 0
2626 0
2627 0
2628 0
2629 0
2630 0
2631 0
2632 0
2633 0
2634 0
2635 0
2636 0
2637 0
2638 0
2639 0
2640 0
2641 0
2642 0
2643 0
2644 0
2645 0
2646 0
2647 0
2648 0
2649 0
2650 0
2651 0
2652 0
2653 0
2654 0
2655 0
2656 0
2657 0
2658 0
2659 0
2660 0
2661 0
2662 0
2663 0
2664 0
2665 0
2666 0
2667 0
2668 0
2669 0
2670 0

MOVED DST ENTRY DEFINITIONS

To support DST continuation records, every continued DST record is merged with its continuation records into one record and is moved to DEBUG permanent memory. A list of moved DST records is kept to allow access to these moved records. Each entry on that list has the following structure:

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	----- MDES_LFLINK -----
1	----- MDES_ORIGINAL_DSTPTR -----
2	----- MDES_REAL_DSTPTR -----
3	----- MDES_REAL_LENGTH -----
4	----- MDES_NEXT_DSTPTR -----

As the RST for a module is built, continuation records are merged and the new "moved" DST pointer is placed in the RST entry. Thus, moved DSTs are transparent for those routines that access the RST.

Those routines that obtain DST pointers from the DST itself are:

- o Routines that deal with SEPTYP records
- o Routines that access indirect type specs
- o Routines that access novel length type specs
- o Routines that access the next DST record

These routines must access those pointers through routines that search the structure to make sure that the DST hasn't been moved. The routines currently provided are:

- o DBG\$RST_DST_PTR - Return the "real" DST pointer
- o DBG\$RST_NEXT_DST - Return the "next" DST pointer

The structure is a singly linked list with its head being DBG\$GL_MOVED_DST_LIST_HEAD.

Define the fields of the Moved DST Record Entry. Also define the entry length and the declaration macro.

```

FIELD MDESFIELDS =
SET
MDES_LFLINK = [ 0, L_ ]; ! Forward link
MDES_ORIGINAL_DSTPTR = [ 1, L_ ]; ! The Original DST pointer

```



```

: 2728 0      MDES_L_REAL_DSTPTR      = [ 2, L_ ],      ! The new, moved, real DST pointer
: 2729 0      MDES_L_REAL_LENGTH      = [ 3, L_ ],      ! The moved DST's length
: 2730 0      MDES_L_NEXT_DSTPTR      = [ 4, L_ ],      ! The next DST pointer
: 2731 0      TES;
: 2732 0
: 2733 0      LITERAL
: 2734 0      MDES_K_LENGTH = 5;          ! Length of an entry in longwords
: 2735 0
: 2736 0      MACRO
: 2737 0      MDES_RECORD = BLOCK[.LONG] FIELD(MDES_FIELDS) %;
: 2738 0
: 2739 0
: 2740 0      ! NOTE: MDES_L_NEXT_DSTPTR is a simple address calculation to the next DST record
: 2741 0      ! with all the continuation records consumed. At the time the calculation is
: 2742 0      ! made it is not known if the record will be relocated. Thus any access to it
: 2743 0      ! must also check it for being relocated by calling DBG$RST_DST_PTR.

```


NUMBER SCANNER STATE TABLE

The Lexical Number Scanner in routine DBGSLEXICAL_SCANNER operates off a Finite-State Machine state table which defines the allowed formats of numeric constants in the current language. There is thus one Number State Table for each language supported by DEBUG (although some languages can share state tables). Each state in the state table is represented by a list of two or more State Table Transition Entries, each of which specifies a state transition to be taken if the current input character is of a class specified in the Transition Entry. The format of a Number Scanner State Table Entry is as follows:

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	First State Table Transition Entry
1	Additional State Table Transition Entries
..	State Table Transition Entry for NUMST\$K_CLASS_OTHER

The individual State Table Transition Entry has the following format:

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	NUMST\$W_NEXT_STATE	NUMST\$B_ACTION	\$B_CHAR_CLASS
---	---------------------	-----------------	----------------

The whole Number Scanner State Table for a language simply consists of a sequence of concatenated State Table Entries, one entry per state in the Finite-State Machine. The whole State Table is thus a vector of State Table Transition Entries. Element zero of the state table is the beginning of the state table's Start State, and each transition's NUMST\$W_NEXT_STATE pointer is a state table index which points to the beginning of the next state's State Table Entry.

A pointer to a Number Scanner State Table is declared as follows:

```
STATE_TABLE: REF NUMST$TABLE
```

This declares STATE_TABLE to be a block vector which is referenced as STATE_TABLE[STATE_INDEX, field-name] where STATE_INDEX is the vector index which points to a specific Number Scanner State Table


```

2801 0 Transition Entry.
2802 0
2803 0
2804 0
2805 0 Define the fields in the Number Scanner State Table Transition Entry.
2806 0 Also define the declaration macro.
2807 0
2808 0 FIELD NUMST$FLD_DEF =
2809 0     SET
2810 0     NUMST$B_CHAR_CLASS = [ 0, B0_ ], Character class that activates
2811 0     NUMST$B_ACTION      = [ 0, B1_ ], this state transition
2812 0     NUMST$W_NEXT_STATE = [ 0, W1_ ], CASE index for semantic action to be
2813 0     TES; taken during state transition
2814 0     Index of next state after transition
2815 0
2816 0
2817 0 MACRO
2818 0     NUMST$TABLE = BLOCKVECTOR[1, LONG] FIELD(NUMST$FLD_DEF) %;
2819 0
2820 0
2821 0 ! Define the allowed values of the character class field NUMST$B_CHAR_CLASS.
2822 0 ! These are the different classes of characters which are significant when
2823 0 ! scanning numeric constants.
2824 0
2825 0 LITERAL
2826 0     NUMST$K_MIN_CLASS      = 0, --- Minimum character class code
2827 0     NUMST$K_CLASS_OTHER   = 0, All other characters
2828 0     NUMST$K_CLASS_DIGIT   = 1, The decimal digits, '0' - '9'
2829 0     NUMST$K_CLASS_HEXDIGIT = 2, The hex digits 'A', 'C', and 'F'
2830 0     NUMST$K_CLASS_DOT     = 3, The decimal point '.'
2831 0     NUMST$K_CLASS_PLUS    = 4, The plus sign '+' (for exponents)
2832 0     NUMST$K_CLASS_MINUS   = 5, The minus sign '-' (for exponents)
2833 0     NUMST$K_CLASS_B       = 6, The letter 'B' (for binary)
2834 0     NUMST$K_CLASS_D       = 7, The letter 'D' (for exponents)
2835 0     NUMST$K_CLASS_E       = 8, The letter 'E' (for exponents)
2836 0     NUMST$K_CLASS_Q       = 9, The letter 'Q' (for exponents)
2837 0     NUMST$K_CLASS_X       = 10, The letter 'X' (for hexadecimal)
2838 0     NUMST$K_CLASS_UNDERSCORE = 11, '.' is allowed in numbers in ADA
2839 0     NUMST$K_CLASS_POUND   = 12, '#' is allowed in numbers in ADA
2840 0     NUMST$K_CLASS_G       = 13, The letter 'G' (for exponents)
2841 0     NUMST$K_MAX_CLASS     = 13; --- Maximum character class code
2842 0
2843 0
2844 0 ! Define the valid values for the NUMST$B_ACTION field. These values are CASE
2845 0 ! indices which select the semantic action routine to be executed for each
2846 0 ! state transition.
2847 0
2848 0 LITERAL
2849 0     NUMST$K_MIN_ACTION     = 1, --- Minimum action index
2850 0     NUMST$K_ACT_DO_NOthing = 1, No semantic action needed
2851 0     NUMST$K_ACT_GO_PAST_DIGIT = 2, Go past digit in integer part
2852 0     NUMST$K_ACT_MARK_DEC_PT = 3, Mark that a decimal point was found
2853 0     NUMST$K_ACT_GO_PAST_FRAC = 4, Go past digit in fraction part
2854 0     NUMST$K_ACT_MARK_E_EXP  = 5, Mark that E-exponent field was found
2855 0     NUMST$K_ACT_MARK_D_EXP  = 6, Mark that D-exponent field was found
2856 0     NUMST$K_ACT_MARK_Q_EXP  = 7, Mark that Q-exponent field was found
2857 0     NUMST$K_ACT_BACKUP_PTRS = 8, Backup pointers and return token

```


:	2858	0	NUMSTSK_ACT_GOT_NUMBER	= 9,	!	We got a number--return token for it
:	2859	0	NUMSTSK_ACT_NOT_NUMBER	= 10,	!	Not a numeric constant
:	2860	0	NUMSTSK_ACT_GIVE_ERROR	= 11,	!	We should never get here--give error
:	2861	0	NUMSTSK_ACT_COB_CKNUM	= 12,	!	In COBOL this could be part of a name
:	2862	0	NUMSTSK_ACT_COB_CKHEX	= 13,	!	In COBOL this could be part of a name
:	2863	0				if it is not in hex mode
:	2864	0	NUMSTSK_ACT_GO_PAST_PACK	= 14,	!	Go past digit in Pack decimal
:	2865	0	NUMSTSK_ACT_GO_PAST_PACK_FRAC	= 15,	!	Go past digit in Pack decimal
:	2866	0	NUMSTSK_ACT_GOT_PACK_NUMBER	= 16,	!	We got a Packed decimal number--
:	2867	0				return token for it
:	2868	0	NUMSTSK_ACT_SAVE_BASE	= 17,	!	base#number in ADA
:	2869	0	NUMSTSK_ACT_MARK_G_EXP	= 18,	!	Mark that G-exponent field was found
:	2870	0	NUMSTSK_MAX_ACTION	= 18;	!	--- Maximum action index

OPERATOR INFORMATION TABLE

The Operator Information Table is a blockvector indexed by Operator Code (e.g., TOKEN\$K_SUBTRACT) which gives pointers to tables which contain the information necessary to select the type conversions and then the routine invocations which evaluate the operators of a given language.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	OPINFO\$L_ROUTTBL
1	OPINFO\$L_HIERTBL
2	OPINFO\$L_INCOMPTBL
3	flags

A pointer to an Operator Information Table is declared as follows:

OPINFO_TABLE: REF OPINFO\$TABLE

Define the fields of the Operator Information Table entry. Also define the declaration macro.

FIELD OPINFO\$FLD_DEF =

SET
OPINFO\$L_ROUTTBL = [0, L_], ! Address of Operator Routine Table
OPINFO\$L_HIERTBL = [1, L_], ! Address of Type Hierarchy Table
OPINFO\$L_INCOMPTBL = [2, L_], ! Address of Type Incompatibility Table
OPINFO\$V_FETCH = [3, V_(0)] ! Flag for whether fetch is done on
this operation.

TES;

MACRO

OPINFO\$TABLE = BLOCKVECTOR[TOKEN\$K_MAX_OPERATOR + 1,4,LONG]
FIELD(OPINFO\$FLD_DEF) %;

OPERATOR ROUTINE TABLE

The Operator Routine Table for a given operator contains one entry for each routine which can be associated with that operator. Each such entry contains the operand types accepted by that routine (e.g., REAL,REAL for F_Floating multiply), the result type of the operation, a routine index which identifies the routine to be invoked, and an optional typeid check routine index which identifies the typeid check routine to be invoked.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	ORT\$W_ROUT	ORT\$B_LEFT_TYPE	ORT\$B_RIGHT_TYPE
1	0	ORT\$B_RESULT_TYPE	ORT\$W_TYPEID_ROUT

A pointer to an operator routine table entry is declared as follows:

ORTPTR : REF ORT\$ENTRY

Define the fields of the Operator Routine Table entry. Also define a declaration macro.

FIELD ORT\$FLD_DEF =

SET			
ORT\$A_BASE_ADDR	= [0, A]	:	Base address
ORT\$B_RIGHT_TYPE	= [0, B0]	:	The right operand Dtype
ORT\$B_LEFT_TYPE	= [0, B1]	:	The left operand Dtype
ORT\$W_TYPES	= [0, W0]	:	The operand Dtype
ORT\$W_ROUT	= [0, W1]	:	The corresponding routine index
ORT\$W_TYPEID_ROUT	= [1, W0]	:	The corresponding type id check routine index
ORT\$B_RESULT_TYPE	= [1, B2]	:	Result Dtype
TES;			

MACRO

ORT\$ENTRY = BLOCK[2, LONG] FIELD(ORT\$FLD_DEF) %;

MACRO

ORT\$TABLE = BLOCKVECTOR[,2,LONG] FIELD(ORT\$FLD_DEF) %;

! This is used to encode "unknown result type" in the dtype field of the result value descriptor constructed in DBGEVALOP.

LITERAL

ORT\$K_RESULT_UNKNOWN = 0;

! Literals that are used in the new style operator evaluation. These

2975 0
2976 0
2977 0
2978 0
2979 0
2980 0
2981 0
2982 0
2983 0
2984 0
2985 0
2986 0
2987 0
2988 0
2989 0
2990 0
2991 0
2992 0
2993 0
2994 0
2995 0
2996 0
2997 0
2998 0
2999 0
3000 0
3001 0
3002 0
3003 0
3004 0
3005 0
3006 0
3007 0
3008 0
3009 0
3010 0
3011 0
3012 0
3013 0
3014 0
3015 0
3016 0
3017 0
3018 0
3019 0
3020 0
3021 0
3022 0
3023 0
3024 0
3025 0
3026 0
3027 0
3028 0
3029 0
3030 0
3031 0

```

: values are stored in the ORTSW_ROUT field of the Operator Routine
: Table entry for a given operator operating on a given pair of
: types. The literal is used as a case index in the DBG$PERFORM_OPERATOR
: routine in order to actually do operations.
:
: *****
: Note - the order of the case indices in DBGPERMOP must correspond
: exactly to the ordering of the literals here. One must be very careful
: in changing this list.
:
: This situation is of course a maintenance problem and we should eventually
: figure out a better way to get these literals shared.
:
: In the meantime, the list below must be kept dense and must correspond
: exactly to the case labels in DBGPERMOP.
: *****
:
: ***
: LITERAL
:
: ORTSK_MIN_ROUT          = 1,
:
:   ! Add.
:   !
:   ORTSK_ADD_B_B          = 1,
:   ORTSK_ADD_BU_BU        = 2,
:   ORTSK_ADD_W_W          = 3,
:   ORTSK_ADD_WU_WU        = 4,
:   ORTSK_ADD_L_L          = 5,
:   ORTSK_ADD_LU_LU        = 6,
:   ORTSK_ADD_F_F          = 7,
:   ORTSK_ADD_D_D          = 8,
:   ORTSK_ADD_G_G          = 9,
:   ORTSK_ADD_H_H          = 10,
:   ORTSK_ADD_FC_FC        = 11,
:   ORTSK_ADD_DC_DC        = 12,
:   ORTSK_ADD_GC_GC        = 13,
:   ORTSK_ADD_HC_HC        = 14,
:
:   ! Subtract.
:   !
:   ORTSK_SUB_B_B          = 15,
:   ORTSK_SUB_BU_BU        = 16,
:   ORTSK_SUB_W_W          = 17,
:   ORTSK_SUB_WU_WU        = 18,
:   ORTSK_SUB_L_L          = 19,
:   ORTSK_SUB_LU_LU        = 20,
:   ORTSK_SUB_F_F          = 21,
:   ORTSK_SUB_D_D          = 22,
:   ORTSK_SUB_G_G          = 23,
:   ORTSK_SUB_H_H          = 24,
:   ORTSK_SUB_FC_FC        = 25,
:   ORTSK_SUB_DC_DC        = 26,
:   ORTSK_SUB_GC_GC        = 27,
:   ORTSK_SUB_HC_HC        = 28,
:
:   ! Divide.
:   !

```



```

3032 0 ORTSK_DIV_B B = 29,
3033 0 ORTSK_DIV_BU BU = 30,
3034 0 ORTSK_DIV_W W = 31,
3035 0 ORTSK_DIV_WU WU = 32,
3036 0 ORTSK_DIV_L L = 33,
3037 0 ORTSK_DIV_LO LU = 34,
3038 0 ORTSK_DIV_F F = 35,
3039 0 ORTSK_DIV_D D = 36,
3040 0 ORTSK_DIV_G G = 37,
3041 0 ORTSK_DIV_H H = 38,
3042 0 ORTSK_DIV_FC FC = 39,
3043 0 ORTSK_DIV_DC DC = 40,
3044 0 ORTSK_DIV_GC GC = 41,
3045 0 ORTSK_DIV_HC HC = 42,
3046 0
3047 0 ! Multiply
3048 0
3049 0 ORTSK_MUL_B B = 43,
3050 0 ORTSK_MUL_BU BU = 44,
3051 0 ORTSK_MUL_W W = 45,
3052 0 ORTSK_MUL_WU WU = 46,
3053 0 ORTSK_MUL_L L = 47,
3054 0 ORTSK_MUL_LO LU = 48,
3055 0 ORTSK_MUL_F F = 49,
3056 0 ORTSK_MUL_D D = 50,
3057 0 ORTSK_MUL_G G = 51,
3058 0 ORTSK_MUL_H H = 52,
3059 0 ORTSK_MUL_FC FC = 53,
3060 0 ORTSK_MUL_DC DC = 54,
3061 0 ORTSK_MUL_GC GC = 55,
3062 0 ORTSK_MUL_HC HC = 56,
3063 0
3064 0 ! Remainder.
3065 0
3066 0 ORTSK_MOD_L L = 57,
3067 0 ORTSK_MOD_LO LU = 58,
3068 0 ORTSK_REM_L L = 59,
3069 0 ORTSK_REM_LO LU = 60,
3070 0
3071 0 ! Shift operations.
3072 0
3073 0 ORTSK_SHIFT_LEFT_L L = 61,
3074 0 ORTSK_SHIFT_RT_L L = 62,
3075 0 ORTSK_SHIFT_RT_LO LU = 63,
3076 0
3077 0 ! Exponentiation.
3078 0
3079 0 ORTSK_POWER_W W = 64,
3080 0 ORTSK_POWER_L L = 65,
3081 0 ORTSK_POWER_F L = 66,
3082 0 ORTSK_POWER_D L = 67,
3083 0 ORTSK_POWER_G L = 68,
3084 0 ORTSK_POWER_H L = 69,
3085 0 ORTSK_POWER_FC L = 70,
3086 0 ORTSK_POWER_DC L = 71,
3087 0 ORTSK_POWER_GC L = 72,
3088 0 ORTSK_POWER_F F = 73,

```


F 1
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 71
(41)

```
3089 0 ORTSK_POWER_D_F = 74,
3090 0 ORTSK_POWER_F_D = 75,
3091 0 ORTSK_POWER_D_D = 76,
3092 0 ORTSK_POWER_G_G = 77,
3093 0 ORTSK_POWER_H_H = 78,
3094 0 ORTSK_POWER_FC_FC = 79,
3095 0 ORTSK_POWER_DC_DC = 80,
3096 0 ORTSK_POWER_GC_GC = 81,
3097 0
3098 0 ! String operations.
3099 0 !
3100 0 ORTSK_CONCAT_T_T = 82,
3101 0 ORTSK_EQL_VT_VT = 83,
3102 0 ORTSK_EQL_T_T = 84,
3103 0 ORTSK_GEQ_VT_VT = 85,
3104 0 ORTSK_GEQ_T_T = 86,
3105 0 ORTSK_GTR_VT_VT = 87,
3106 0 ORTSK_GTR_T_T = 88,
3107 0 ORTSK_LEQ_VT_VT = 89,
3108 0 ORTSK_LEQ_T_T = 90,
3109 0 ORTSK_LSS_VT_VT = 91,
3110 0 ORTSK_LSS_T_T = 92,
3111 0 ORTSK_NEQ_VT_VT = 93,
3112 0 ORTSK_NEQ_T_T = 94,
3113 0
3114 0 ! Relational.
3115 0 !
3116 0 ORTSK_EQL_B_B = 95,
3117 0 ORTSK_EQL_W_W = 96,
3118 0 ORTSK_EQL_L_L = 97,
3119 0 ORTSK_EQL_F_F = 98,
3120 0 ORTSK_EQL_D_D = 99,
3121 0 ORTSK_EQL_G_G = 100,
3122 0 ORTSK_EQL_H_H = 101,
3123 0 ORTSK_EQL_FC_FC = 102,
3124 0 ORTSK_EQL_DC_DC = 103,
3125 0 ORTSK_EQL_GC_GC = 104,
3126 0 ORTSK_EQL_HC_HC = 105,
3127 0
3128 0 ORTSK_NEQ_B_B = 106,
3129 0 ORTSK_NEQ_W_W = 107,
3130 0 ORTSK_NEQ_L_L = 108,
3131 0 ORTSK_NEQ_F_F = 109,
3132 0 ORTSK_NEQ_D_D = 110,
3133 0 ORTSK_NEQ_G_G = 111,
3134 0 ORTSK_NEQ_H_H = 112,
3135 0 ORTSK_NEQ_FC_FC = 113,
3136 0 ORTSK_NEQ_DC_DC = 114,
3137 0 ORTSK_NEQ_GC_GC = 115,
3138 0 ORTSK_NEQ_HC_HC = 116,
3139 0
3140 0 ORTSK_GEQ_B_B = 117,
3141 0 ORTSK_GEQ_W_W = 118,
3142 0 ORTSK_GEQ_L_L = 119,
3143 0 ORTSK_GEQ_LD_LU = 120,
3144 0 ORTSK_GEQ_F_F = 121,
3145 0 ORTSK_GEQ_D_D = 122,
```


G 1
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 72
(41)

```
3146 0 ORTSK_GEQ_G_G = 123.  
3147 0 ORTSK_GEQ_H_H = 124.  
3148 0  
3149 0 ORTSK_GTR_B_B = 125.  
3150 0 ORTSK_GTR_W_W = 126.  
3151 0 ORTSK_GTR_L_L = 127.  
3152 0 ORTSK_GTR_LO_LU = 128.  
3153 0 ORTSK_GTR_F_F = 129.  
3154 0 ORTSK_GTR_D_D = 130.  
3155 0 ORTSK_GTR_G_G = 131.  
3156 0 ORTSK_GTR_H_H = 132.  
3157 0  
3158 0 ORTSK_LEQ_B_B = 133.  
3159 0 ORTSK_LEQ_W_W = 134.  
3160 0 ORTSK_LEQ_L_L = 135.  
3161 0 ORTSK_LEQ_LO_LU = 136.  
3162 0 ORTSK_LEQ_F_F = 137.  
3163 0 ORTSK_LEQ_D_D = 138.  
3164 0 ORTSK_LEQ_G_G = 139.  
3165 0 ORTSK_LEQ_H_H = 140.  
3166 0  
3167 0 ORTSK_LSS_B_B = 141.  
3168 0 ORTSK_LSS_W_W = 142.  
3169 0 ORTSK_LSS_L_L = 143.  
3170 0 ORTSK_LSS_LO_LU = 144.  
3171 0 ORTSK_LSS_F_F = 145.  
3172 0 ORTSK_LSS_D_D = 146.  
3173 0 ORTSK_LSS_G_G = 147.  
3174 0 ORTSK_LSS_H_H = 148.  
3175 0  
3176 0 ! Bitwise logical operations.  
3177 0 !  
3178 0 ORTSK_BIT_AND_B_B = 149.  
3179 0 ORTSK_BIT_AND_W_W = 150.  
3180 0 ORTSK_BIT_AND_L_L = 151.  
3181 0 ORTSK_BIT_EQV_B_B = 152.  
3182 0 ORTSK_BIT_EQV_W_W = 153.  
3183 0 ORTSK_BIT_EQV_L_L = 154.  
3184 0 ORTSK_BIT_NOT_B = 155.  
3185 0 ORTSK_BIT_NOT_W = 156.  
3186 0 ORTSK_BIT_NOT_L = 157.  
3187 0 ORTSK_BIT_OR_B_B = 158.  
3188 0 ORTSK_BIT_OR_W_W = 159.  
3189 0 ORTSK_BIT_OR_L_L = 160.  
3190 0 ORTSK_BIT_XOR_B_B = 161.  
3191 0 ORTSK_BIT_XOR_W_W = 162.  
3192 0 ORTSK_BIT_XOR_L_L = 163.  
3193 0  
3194 0 ! Logical operations.  
3195 0 !  
3196 0 ORTSK_AND_B_B = 164.  
3197 0 ORTSK_AND_W_W = 165.  
3198 0 ORTSK_AND_L_L = 166.  
3199 0 ORTSK_AND_D_D = 167.  
3200 0 ORTSK_NOT_B = 168.  
3201 0 ORTSK_NOT_W = 169.  
3202 0 ORTSK_NOT_L = 170.
```


H 1
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 73
(41)

```
3203 0 ORTSK_NOT_D = 171,
3204 0 ORTSK_OR_B_B = 172,
3205 0 ORTSK_OR_W_W = 173,
3206 0 ORTSK_OR_L_L = 174,
3207 0 ORTSK_OR_D_D = 175,
3208 0 ORTSK_XOR_L_L = 176,
3209 0
3210 0 ! Unary plus and minus.
3211 0 !
3212 0 ORTSK_UNARY_MINUS_B = 177,
3213 0 ORTSK_UNARY_MINUS_W = 178,
3214 0 ORTSK_UNARY_MINUS_L = 179,
3215 0 ORTSK_UNARY_MINUS_LU = 180,
3216 0 ORTSK_UNARY_MINUS_F = 181,
3217 0 ORTSK_UNARY_MINUS_D = 182,
3218 0 ORTSK_UNARY_MINUS_G = 183,
3219 0 ORTSK_UNARY_MINUS_H = 184,
3220 0 ORTSK_UNARY_MINUS_FC = 185,
3221 0 ORTSK_UNARY_MINUS_DC = 186,
3222 0 ORTSK_UNARY_MINUS_GC = 187,
3223 0 ORTSK_UNARY_MINUS_HC = 188,
3224 0 ORTSK_UNARY_PLUS_B = 189,
3225 0 ORTSK_UNARY_PLUS_W = 190,
3226 0 ORTSK_UNARY_PLUS_L = 191,
3227 0 ORTSK_UNARY_PLUS_F = 192,
3228 0 ORTSK_UNARY_PLUS_D = 193,
3229 0 ORTSK_UNARY_PLUS_G = 194,
3230 0 ORTSK_UNARY_PLUS_H = 195,
3231 0 ORTSK_UNARY_PLUS_FC = 196,
3232 0 ORTSK_UNARY_PLUS_DC = 197,
3233 0 ORTSK_UNARY_PLUS_GC = 198,
3234 0 ORTSK_UNARY_PLUS_HC = 199,
3235 0
3236 0 ! Absolute Value.
3237 0 !
3238 0 ORTSK_ABS_B = 200,
3239 0 ORTSK_ABS_W = 201,
3240 0 ORTSK_ABS_L = 202,
3241 0 ORTSK_ABS_F = 203,
3242 0 ORTSK_ABS_D = 204,
3243 0 ORTSK_ABS_G = 205,
3244 0 ORTSK_ABS_H = 206,
3245 0
3246 0 ! Indirection.
3247 0 !
3248 0 ORTSK_INDIRECT_LU = 207,
3249 0 ORTSK_INDIRECT_TPTR = 208,
3250 0
3251 0 ! BLISS bit-select.
3252 0 !
3253 0 ORTSK_BITSELECT = 209,
3254 0
3255 0 ! Set operations.
3256 0 !
3257 0 ORTSK_DIFFERENCE_SET_SET = 210,
3258 0 ORTSK_EQ_L_SET_SET = 211,
3259 0 ORTSK_GEQ_SET_SET = 212,
```


I 1
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 74
(41)

```
3260 0 ORTSK_IN_SET_SET = 213,
3261 0 ORTSK_INTERSECT_SET_SET = 214,
3262 0 ORTSK_LEQ_SET_SET = 215,
3263 0 ORTSK_NEQ_SET_SET = 216,
3264 0 ORTSK_UNION_SET_SET = 217,
3265 0
3266 0 ! C operations:
3267 0 ! ADDRESS_OF (&), SIZEOF
3268 0 ! Also, addition or subtraction involving typed pointers
3269 0 ! is special-cased in C because there is scaling to be done.
3270 0
3271 0 ORTSK_ADDRESS_L = 218,
3272 0 ORTSK_SIZEOF_C = 219,
3273 0 ORTSK_ADD_TPTR_L = 220,
3274 0 ORTSK_SUB_TPTR_L = 221,
3275 0 ORTSK_SUB_TPTR_TPTR = 222,
3276 0
3277 0 ! BASIC implication (A imp B)
3278 0 !
3279 0 ORTSK_BIT_IMP_B_B = 223,
3280 0 ORTSK_BIT_IMP_W_W = 224,
3281 0 ORTSK_BIT_IMP_L_L = 225,
3282 0
3283 0 ! C pre- and post- increment, decrement (++X X++ --X X--)
3284 0 !
3285 0 ORTSK_PRE_INCR_L = 226,
3286 0 ORTSK_PRE_INCR_LU = 227,
3287 0 ORTSK_PRE_INCR_D = 228,
3288 0 ORTSK_PRE_INCR_TPTR = 229,
3289 0 ORTSK_POST_INCR_L = 230,
3290 0 ORTSK_POST_INCR_LU = 231,
3291 0 ORTSK_POST_INCR_D = 232,
3292 0 ORTSK_POST_INCR_TPTR = 233,
3293 0 ORTSK_PRE_DECR_C = 234,
3294 0 ORTSK_PRE_DECR_LU = 235,
3295 0 ORTSK_PRE_DECR_D = 236,
3296 0 ORTSK_PRE_DECR_TPTR = 237,
3297 0 ORTSK_POST_DECR_L = 238,
3298 0 ORTSK_POST_DECR_LU = 239,
3299 0 ORTSK_POST_DECR_D = 240,
3300 0 ORTSK_POST_DECR_TPTR = 241,
3301 0
3302 0 ! Packed Decimal Operations.
3303 0 !
3304 0 ORTSK_ADD_P_P = 242,
3305 0 ORTSK_SUB_P_P = 243,
3306 0 ORTSK_MUL_P_P = 244,
3307 0 ORTSK_DIV_P_P = 245,
3308 0 ORTSK_UNARY_PLUS_P = 246,
3309 0 ORTSK_UNARY_MINUS_P = 247,
3310 0 ORTSK_EQ_P_P = 248,
3311 0 ORTSK_NEQ_P_P = 249,
3312 0 ORTSK_GTR_P_P = 250,
3313 0 ORTSK_GEQ_P_P = 251,
3314 0 ORTSK_LSS_P_P = 252,
3315 0 ORTSK_LEQ_P_P = 253,
3316 0
```



```

3317 0      ! PL/I Bit-String Operations.
3318 0      !
3319 0      ORTSK_CONCAT_TF_TF      = 254,
3320 0      ORTSK_EQ_TF_TF        = 255,
3321 0      ORTSK_NEQ_TF_TF       = 256,
3322 0      ORTSK_GTR_TF_TF       = 257,
3323 0      ORTSK_GEQ_TF_TF       = 258,
3324 0      ORTSK_LSS_TF_TF       = 259,
3325 0      ORTSK_LEQ_TF_TF       = 260,
3326 0      ORTSK_BIT_NOT_TF      = 261,
3327 0      ORTSK_BIT_AND_TF      = 262,
3328 0      ORTSK_BIT_OR_TF       = 263,
3329 0
3330 0      ORTSK_EQL_RFA_RFA      = 264,
3331 0      ORTSK_NEQ_RFA_RFA      = 265,
3332 0
3333 0      ORTSK_UNARY_PLUS_Q      = 266,
3334 0      ORTSK_UNARY_MINUS_Q    = 267,
3335 0      ORTSK_UNARY_PLUS_O     = 268,
3336 0      ORTSK_UNARY_MINUS_O    = 269,
3337 0
3338 0      ! Built-in Functions
3339 0      !
3340 0      ORTSK_SUCC_ENUM        = 270,
3341 0      ORTSK_PRED_ENUM       = 271,
3342 0
3343 0      ! Fixed binary operations.
3344 0      !
3345 0      ORTSK_UNARY_PLUS_FIXED = 272,
3346 0      ORTSK_UNARY_MINUS_FIXED = 273,
3347 0      ORTSK_ABS_FIXED       = 274,
3348 0      ORTSK_ADD_FIXED_FIXED = 275,
3349 0      ORTSK_SUB_FIXED_FIXED = 276,
3350 0      ORTSK_MUL_FIXED_FIXED = 277,
3351 0      ORTSK_DIV_FIXED_FIXED = 278,
3352 0      ORTSK_EQL_FIXED_FIXED = 279,
3353 0      ORTSK_NEQ_FIXED_FIXED = 280,
3354 0      ORTSK_LSS_FIXED_FIXED = 281,
3355 0      ORTSK_GTR_FIXED_FIXED = 282,
3356 0      ORTSK_LEQ_FIXED_FIXED = 283,
3357 0      ORTSK_GEQ_FIXED_FIXED = 284,
3358 0      ORTSK_MAX_ROUT        = 284;
3359 0
3360 0
3361 0
3362 0
3363 0      ! Literals that are defined for type id check in DBG$PERFORM_TYPEID_CHECK.
3364 0      !
3365 0      LITERAL
3366 0      ORTSK_TYPEID_MIN_ROUT   = 1,
3367 0      ORTSK_TYPEID_ENUM_ENUM = 1,
3368 0      ORTSK_TYPEID_SET_SET    = 2,
3369 0      ORTSK_TYPEID_TPTR_TPTR  = 3,
3370 0      ORTSK_TYPEID_SUBRNG_SUBRNG = 4,
3371 0      ORTSK_TYPEID_MAX_ROUT   = 4;
3372 0
3373 0

```

!*** ALSO CHANGE the corresponding
!*** definition in DBGPERMOP.MAR if
!*** you change this.

! Literals that are defined for type id check in DBG\$PERFORM_TYPEID_CHECK.

LITERAL

```

ORTSK_TYPEID_MIN_ROUT   = 1,
ORTSK_TYPEID_ENUM_ENUM = 1,
ORTSK_TYPEID_SET_SET    = 2,
ORTSK_TYPEID_TPTR_TPTR  = 3,
ORTSK_TYPEID_SUBRNG_SUBRNG = 4,
ORTSK_TYPEID_MAX_ROUT   = 4;

```

!***

TYPE GRAPH

The type conversions that are done for a given operator in a given language are specified by a set of directed acyclic graphs. For example, part of the graph that specifies the implicit type conversions to be done for FORTRAN addition might look like:

```

L -> F -> D -> H
      \   \
      FC -> DC

```

Each graph is represented by listing its edges. Each edge is just a pair of Type IDs.

Below we define the data structure used to declare the edges of these graphs.

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

```

+-----+
| HIGHER_TYPE | LOWER_TYPE |
+-----+

```

Define the fields in an entry in an edge of a Type Graph.

```

FIELD TYPE_GRAPH$FLD_DEF =
SET
TYPE_GRAPH$B_LOWER_TYPE = [0,B0_],
TYPE_GRAPH$B_HIGHER_TYPE = [0,B1_],
TYPE_GRAPH$W_BOTH_TYPES = [0,W0_]
TES;

```

! Define the macro which is used to declare an entry in a Type Graph

```

MACRO
TYPE_GRAPH$ENTRY = BLOCK [1,WORD] FIELD (TYPE_GRAPH$FLD_DEF) %;

```


TYPE CONVERSION INFORMATION TABLE

The Type Conversion Information Table is a vector of longwords used by a language which gives pointers to tables which contain the information necessary to select the type conversions and then the routine invocations which does the type conversion.

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	CVTINFO\$MAPTBL
1	CVTINFO\$CVTTBL
2	flags

A pointer to an Type Conversion Information Table is declared as follows:

CVTINFO_TABLE: REF CVTINFO\$TABLE

Define the fields of the Type Conversion Information Table entry. Also define the declaration macro.

```
FIELD CVTINFO$FLD_DEF =
SET
CVTINFO$MAPTBL = [ 0, L_ ], ! Address of Type Map Table
CVTINFO$CVTTBL = [ 1, L_ ], ! Address of Type Conversion Table
CVTINFO$V_ROUND = [ 2, V_(0) ], ! Flag to indicate conversion result
                                ! is rounded
TES;
```

```
MACRO
CVTINFO$TABLE = BLOCK[3, LONG] FIELD (CVTINFO$FLD_DEF) %;
```


TYPE CONVERT ENTRY

A Type Convert Entry contains a type conversion routine index which identifies the routine to be invoked, from data type, and to data type.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

+-----+
| TYPE_CVT$W_ROUT      | HIGHER_TYPE | LOWER_TYPE |
+-----+

```

A pointer to an Type Map Entry is declared as follows:

CVTPTR : REF TYPE_CVT\$ENTRY

Define the fields of the Type Map Table entry. Also define an entry declaration macro.

FIELD

```

TYPE_CVT$FLD_DEF =
SET
TYPE_CVT$B_LOWER_TYPE = [0, B0_],
TYPE_CVT$B_HIGHER_TYPE = [0, B1_],
TYPE_CVT$W_MAP_PAIR = [0, W0_],
TYPE_CVT$W_ROUT = [0, W1_]
TES;

```

MACRO

TYPE_CVT\$ENTRY = BLOCK[1, LONG] FIELD (TYPE_CVT\$FLD_DEF) %;

! Literals that are used in the new style operator evaluation.
! The literal is used as a case index in the DBG\$LANGUAGE_TYPE_CONV routine in order to actually do the conversion.

!***

LITERAL

```

CVT$K_MIN_ROUT      = 1,
CVT$K_PLI_CVT       = 1,
CVT$K_COB_PICT      = 2,
CVT$K_MAX_ROUT      = 2;

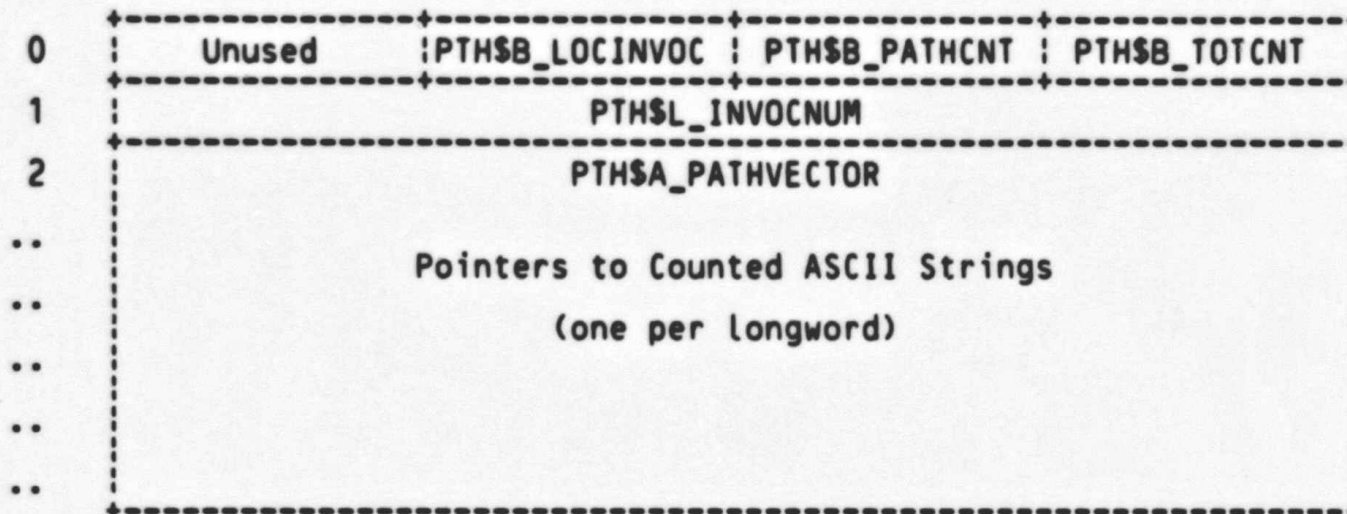
```

!***

PATHNAME DESCRIPTORS

The pathname descriptor is used internally in the Debugger to represent a pathname, including data qualification. Thus 'MOD\ROUT\X' is a pathname in this sense, and so is 'MOD\ROUT\A.B.C'. The format of a pathname descriptor is shown here:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



The Pathname Descriptor contains two counts fields which give the total number of names in the pathname (five for M\R\A.B.C) and the total number of names before the first dot (three for M\R\A.B.C). It also contains an invocation number (which is zero if none was explicitly specified) and an index into PTH\$A_PATHVECTOR which indicates which pathname component was qualified by the invocation number (1 for A 2\X and 4 for M\R1\R2\R3 5\X). This index is zero if no invocation number was given. The rest of the descriptor consists of longwords containing pointers to the individual names in the pathname, each represented by a Counted ASCII String.

A pointer to a Pathname Descriptor is declared as follows:

PATHPTR: REF PTH\$PATHNAME

Define the Pathname Descriptor fields.

FIELD PTH\$FLD_DEF =

SET

PTH\$B_TOTCNT = [0, 80_], ; Total number of names in pathname
including all data qualification


```

: 3564 0      PTH$B_PATHCNT = [ 0, B1_ ],      ! Number of names in pathname proper
: 3565 0      PTH$B_LOCINVOC = [ 0, B2_ ],      ! before any data qualification
: 3566 0      ! --- = [ 0, B3_ ],              ! The location in the vector below of
: 3567 0      PTH$L_INVOCNUM = [ 1, L_ ],        ! the name with an invocation num-
: 3568 0      PTH$A_PATHVECTOR = [ 2, A_ ],      ! ber (first name is 1) or zero
: 3569 0      ! Unused
: 3570 0      ! The invocation number or zero
: 3571 0      ! The start of the pathname vector--has
: 3572 0      ! one name pointer (pointing to a
: 3573 0      ! Counted ASCII name) per longword.
: 3574 0      TES;
: 3575 0
: 3576 0      ! Define the maximum allowed size of a pathname, i.e. the maximum number of
: 3577 0      ! individual names, including data components. Also define the size of the
: 3578 0      ! fixed part of the descriptor and the declaration macro.
: 3579 0
: 3580 0      LITERAL
: 3581 0      DBG$K_MAX_PATHNAME = 50,          ! Maximum size of a pathname
: 3582 0      DBG$K_PATHDESCSIZE = 2,          ! Size of fixed part of Pathname
: 3583 0      ! Descriptor
: 3584 0      DBG$K_PATHNAME_SIZE = DBG$K_MAX_PATHNAME + DBG$K_PATHDESCSIZE;
: 3585 0
: 3586 0      MACRO
: 3587 0      PTH$PATHNAME = BLOCK[DBG$K_PATHNAME_SIZE] FIELD(PTH$FLD_DEF) %;
: 3588 0      !**~
: 3589 0

```


PERMANENT SYMBOL DESCRIPTORS

The Permanent Symbol Descriptor is used to describe Debugger "permanent symbols" in Value Descriptors, Primary Descriptors, and address expressions. Permanent symbols name the VAX registers; "R0" is thus a permanent symbol. Permanent Symbol Descriptors are only used for languages COBOL and PL/I. A Permanent Symbol Descriptor has this format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	Unused	DBG\$B_PERMSD_ID
1	DBG\$L_PERMSD_ADDR	
2	DBG\$L_PERMSD_PATH	

A pointer to a Permanent Symbol Descriptor is declared as follows:

PSDPTR: REF DBG\$PERMSD

Define the Permanent Symbol Descriptor fields and declaration macro.

FIELD DBG\$PERMSD_FIELDS =

SET
DBG\$B_PERMSD_ID = [0, B0_], ! Symbol ID (register number)
DBG\$L_PERMSD_ADDR = [1, L_], ! Address of the register contents
DBG\$L_PERMSD_PATH = [2, L_], ! Address of symbol's pathname descr.
TES;

LITERAL

DBG\$K_PERMSD_SIZE = 3; ! Size of Permanent Symbol Descriptor
! in longwords

MACRO

DBG\$PERMSD = BLOCK[DBG\$K_PERMSD_SIZE] FIELD(DBG\$PERMSD_FIELDS) %;

! These are the possible values of the DBG\$B_PERMSD_ID field.

LITERAL

DBG\$K_R0 = 200, ! Register R0
DBG\$K_R1 = 201, ! Register R1
DBG\$K_R2 = 202, ! Register R2
DBG\$K_R3 = 203, ! Register R3
DBG\$K_R4 = 204, ! Register R4
DBG\$K_R5 = 205, ! Register R5
DBG\$K_R6 = 206, ! Register R6

D 2
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 82
(46)

:	3647	0		DBG\$K_R7	=	207,	:	Register R7
:	3648	0		DBG\$K_R8	=	208,	:	Register R8
:	3649	0		DBG\$K_R9	=	209,	:	Register R9
:	3650	0		DBG\$K_R10	=	210,	:	Register R10
:	3651	0		DBG\$K_R11	=	211,	:	Register R11
:	3652	0		DBG\$K_AP	=	212,	:	Argument Pointer (AP)
:	3653	0		DBG\$K_FP	=	213,	:	Frame Pointer (FP)
:	3654	0		DBG\$K_SP	=	214,	:	Stack Pointer (SP)
:	3655	0		DBG\$K_PC	=	215,	:	Program Counter (PC)
:	3656	0		DBG\$K_PSL	=	216,	:	Processor Status Longword (PSL)
:	3657	0	!***-					

PREDECLARED IDENTIFIER ENTRY

Language reserves Predeclared Identifier as names of functions, procedures, types, values, etc. Predeclared Identifier Entry is used by DEBUG to describe one of these predeclared identifiers. An identifier is looked up in this table, if an entry is found, a value descriptor is created for this entry. For example, TRUE, FALSE in PASCAL, .TRUE., .FALSE., in FORTRAN. The Predeclared Identifier Entry has the following format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	unused	PRIDSB_FCODE	PRIDSB_DTYPE	PRIDSB_KIND
1	PRIDSL_VALUE			
2		PRIDSA_NAME	PRIDSB_LENGTH	
..	(Predeclared Identifier name in Counted ASCII)			
..	(Includes PRIDSB_LENGTH and PRIDSA_NAME fields)			
..				

The PRIDSB_KIND field defines the kind of Predefined Identifier. The PRIDSB_KIND field of an Predefined Identifier Constant Entry always has the value of PRIDSK_CONSTANT. It has the following format.

A pointer to a Predefined Identifier Value Entry is defined as follows:

PRIDPTR: REF PRIDSEENTRY

Define the fields of the Predefined Identifier Entry.

FIELD PRID\$FLD_DEF =

SET		
PRIDSB_KIND	= [0, B0_]	! Predefined Identifier Kind
PRIDSB_DTYPE	= [0, B1_]	! Data Type of the Predefined ID constant
PRIDSB_FCODE	= [0, B2_]	! FCODE of the data type
PRIDSL_VALUE	= [1, L_]	! Data value of the Predefined ID constant
PRIDSB_LENGTH	= [2, B0_]	! Length of the Predefined ID name in bytes
PRIDSA_NAME	= [2, A1_]	! Predefined ID name in ASCII
TES;		

LITERAL

PRIDSK_ENTSIZE = 2; ! Size of fixed part is 2 longwords

MACRO

PRIDSEENTRY = BLOCK[,LONG] FIELD(PRID\$FLD_DEF) %;

F 2
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 84
(47)

3715 0
3716 00
3717 00
3718 00
3719 00
3720 0

! Define the valid values of the PRIDSB_KIND field.

LITERAL

PRIDSK_CONSTANT = 1;

! Predeclared ID constant

PRIMARY AND VALUE DESCRIPTOR HEADER

Primary Descriptors and Value Descriptors are descriptors built by DEBUG to describe Primary Symbols and language values, respectively. The language-independent versions of these descriptors all have a common header section which has the following format:

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	SB_DHDR_LANG	SB_DHDR_TYPE	DBG\$W_DHDR_LENGTH
1	SB_DHDR_KIND	SB_DHDR_FCODE	DBG\$W_DHDR_FLAGS
2	DBG\$L_DHDR_TYPEID		
3	DBG\$L_DHDR_SYMIDO		

Define the Primary or Value Descriptor Header Block fields.

FIELD DBG\$DHDR_FIELDS =

```

SET
DBG$B_DHDR_LANG      = [ 0, B3 ],      Language Code
DBG$B_DHDR_TYPE      = [ 0, B2 ],      Type Code (Primary/Value/...)
DBG$W_DHDR_LENGTH    = [ 0, W0 ],      Length of descriptor in bytes
DBG$B_DHDR_KIND      = [ 1, B3 ],      RST kind of entity
DBG$B_DHDR_FCODE     = [ 1, B2 ],      FCODE of data item
DBG$W_DHDR_FLAGS     = [ 1, W0 ],      Flags word
DBG$V_DHDR_AGGR      = [ 1, V0(0) ],    Set if data item is an aggregate
DBG$V_DHDR_SUBREF    = [ 1, V0(1) ],    Set if partial data reference
DBG$V_DHDR_BITREF    = [ 1, V0(2) ],    Set if bit-string sub-reference
DBG$V_DHDR_SGNEXT    = [ 1, V0(3) ],    Set if value should be sign-extended
DBG$V_DHDR_BLIBLK    = [ 1, V0(4) ],    Set if data item is a BLISS block
DBG$V_DHDR_UNCVT     = [ 1, V0(5) ],    Unconverted constant flag
DBG$V_DHDR_LITERAL   = [ 1, V0(6) ],    Flag for literal value
DBG$V_DHDR_OVERRIDE  = [ 1, V0(7) ],    Set if override was present
DBG$V_DHDR_TMPREF    = [ 1, V1(0) ],    Set if temporary SUBREF
DBG$V_DHDR_FORMAT    = [ 1, V1(4,4) ],  Printing Format Code
DBG$L_DHDR_TYPEID    = [ 2, L ],        TYPEID of data item
DBG$L_DHDR_SYMIDO    = [ 3, L ],        Context Symid (root data item)
TES;

```


The Primary Descriptor and the Value Descriptor for COBOL and PL/I are descriptor blocks built by the language-specific routines for those two languages. As such their contents are unknown to the Debugger kernel (and are not described here), but they all must have the common header described here. A Primary Descriptor or Value Descriptor for COBOL or PL/I thus has this format:

! The possible values of the DBGSB_DHEAD_LANG field are the language codes


```
: 3826 0      ! defined in the section entitled 'Miscellaneous Literals' above.  The possible
: 3827 0      ! value of the DBG$B_DHEAD_TYPE field, which are defined in the same section,
: 3828 0      ! are the following:
: 3829 0
: 3830 0      !     DBG$K_PRIMARY_DESC      ! This is a Primary Descriptor
: 3831 0      !     DBG$K_VALUE_DESC      ! This is a non-volatile Value Descriptor
: 3832 0      !     DBG$K_V_VALUE_DESC      ! This is a volatile Value Descriptor
: 3833 0      ! **-
```


PRIMARY DESCRIPTOR DEFINITIONS

Primary Descriptors describe Primary Symbols parsed in DEBUG commands. A Primary Symbol may be a simple name, such as "ABC", or a more complex name including pathname qualification, data qualification, subscripting, and dereferencing, such as "MOD\ROUT\A.B[2,3]^C(4).D".

The Primary Descriptor described here is the Primary Descriptor built by the language-independent parser in module DBGPARSER.

A Primary Descriptor consists of a Root Node followed by a doubly linked list of Sub-Nodes which describe the individual components of the symbol described by the Primary Descriptor. Each Sub-Node thus describes an instance of data qualification, subscripting, dereferencing, or the like. The last Sub-Node describes the final object named by the Primary Symbol.

Consider the Primary Symbol "A\B\C.D[2]^E" for example. The Primary Descriptor Root Node would point to a Sub-Node for A\B\C which requires a Record Sub-Node. It points to another Sub-Node for component D of record A\B\C. This component is an array and requires an Array Sub-Node. The next node is the Sub-Node for the array element A\B\C.D[2]. This element is of a pointer type. The next Sub-Node is a node for the pointed-to object which happens to be a record and therefore calls for a Record Sub-Node. Finally, the last Sub-Node on the chain is for component E of that record. The type of Sub-Node required depends on the data type of E, but would be a Normal Sub-Node for all data types other than arrays and records.

```

3834 0
3835 0
3836 0
3837 0
3838 0
3839 0
3840 0
3841 0
3842 0
3843 0
3844 0
3845 0
3846 0
3847 0
3848 0
3849 0
3850 0
3851 0
3852 0
3853 0
3854 0
3855 0
3856 0
3857 0
3858 0
3859 0
3860 0
3861 0
3862 0
3863 0

```


A Primary Descriptor consists of a Root Node followed by a doubly linked list of Sub-Nodes which describe the individual components of the symbol described by the Primary Descriptor. The format of the Primary Descriptor Root Node is as follows:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0

0	\$B_DHDR_LANG	\$B_DHDR_TYPE	DBG\$W_DHDR_LENGTH
1	\$B_DHDR_KIND	\$B_DHDR_FCODE	DBG\$W_DHDR_FLAGS
2	DBG\$L_DHDR_TYPEID		
3	DBG\$L_DHDR_SYMIDO		
4	DBG\$W_PRIM_LENGTH	DBG\$W_PRIM_OFFSET	
5	DBG\$L_PRIM_FLINK		
6	DBG\$L_PRIM_BLINK		
7	Unused		_SCOPE_STATE
8	DBG\$L_PRIM_SCOPE		

PRIMPTR: REF DBG\$PRIMARY

Define the fields of the language-independent Primary Descriptor Root Node.

FIELD DBG\$PRIM_FIELDS =

```

SET
DBG$W_PRIM_OFFSET      = [ 4, SW0_ ], ! Offset within data item
DBG$W_PRIM_LENGTH      = [ 4, W1_ ], ! Length of partial reference
DBG$A_PRIM_FLINK       = [ 5, A_ ], ! Address of Sub-Node list head in root
DBG$L_PRIM_FLINK       = [ 5, L_ ], ! Forward link to first Sub-Node
DBG$L_PRIM_BLINK       = [ 6, L_ ], ! Backward link to last Sub-Node
DBG$B_PRIM_SCOPE_STATE = [ 7, B0_ ], ! Scope state (see next field)
DBG$L_PRIM_SCOPE       = [ 8, L_ ], ! Scope where symbol was looked up
TES;

```

```
! Define the declaration macro. Note that the Primary Descriptor also has the
! Value Descriptor fields defined. This is to simplify the declarations in
! those routines which can accept either a Primary Descriptor or a Value
```


L 2
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 90
(51)

```

: 3921 0      ! Descriptor as input.
: 3922 0
: 3923 0
M 3924 0      MACRO
: 3925 0      DBG$PRIMARY = BLOCK [,LONG]
: 3926 0      FIELD(DBG$DHDR_FIELDS, DBG$PRIM_FIELDS, DBG$VALUE_FIELDS) %;
: 3927 0
: 3928 0      ! Define the size in longwords of the Primary Descriptor Root Node.
: 3929 0
: 3930 0      LITERAL
: 3931 0      DBG$K_PRIMARY_SIZE      = 9;      ! Size of Root Node in longwords
```


Primary Descriptor Sub-Nodes are attached to the Primary Descriptor Root Node via a doubly linked list. The first Sub-Node represents the first data item in the Primary Symbol (for example, A/B in "A/B.C[2].D") and subsequent nodes represent subsequent items (such as .C, .C[2], and .D). All Primary Descriptor Sub-Nodes include a "common core" of information, with other fields added for certain specific sub-nodes. The "Normal Sub-Node" described here is used for all "normal" symbols, meaning all symbols for which no specific Primary Descriptor Sub-Node has been defined below. This Sub-Node has only the "common core" fields and no type-specific fields:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0

0	DBG\$PNODE_FLINK			
1	DBG\$PNODE_BLINK			
2	Unused	!\$B_PNODE_FLAGS	!\$B_PNODE_FCODE	!\$B_PNODE_KIND
3	DBG\$PNODE_TYPEID			
4	DBG\$PNODE_SYMID			
5	DBG\$PNODE_RELOC			

PNODEPTR: REF DBG\$PRIM_NODE

The following are the meanings of the fields in the Normal Sub-Node.

[illegible]


```

3989 0
3990 0
3991 0
3992 0
3993 0
3994 0
3995 0
3996 0
3997 0
3998 0
3999 0
4000 0
4001 0
4002 0
4003 0
4004 0
4005 0
4006 0
4007 0
4008 0
4009 0
4010 0
4011 0
4012 0
4013 0
4014 0
4015 0
4016 0
4017 0
4018 0
4019 0
4020 0
4021 0
4022 0
4023 0
4024 0
4025 0
4026 0
4027 0
4028 0
4029 0
4030 0
4031 0
4032 0
4033 0
4034 0
4035 0
4036 0
4037 0
4038 0
4039 0
4040 0
4041 0
4042 0
4043 0

```

! Define the size of the Primary Descriptor Normal Sub-Node.

LITERAL

DBG\$K_PRIM_SIZE_NORMAL = 6; ! Size of Normal Sub-Node in longwords

! Define the fields of the Primary Descriptor Sub-Node. Also define the declaration macro.

FIELD DBG\$PNODE_FIELDS =

SET

DBG\$P_PNODE_FLINK = [0, L_], ! Link to following primary sub-node

DBG\$P_PNODE_BLINK = [1, L_], ! Link to preceding primary sub-node

DBG\$B_PNODE_KIND = [2, B0_], ! The RST "kind" of this component

DBG\$B_PNODE_FCODE = [2, B1_], ! FCODE for this component

DBG\$B_PNODE_FLAGS = [2, B2_], ! Flags byte

DBG\$V_PNODE_EVAL = [2, V2_(0)], ! Evaluate subscripting, dot qualification, or dereferencing operation on this component

DBG\$V_PNARR_COLUMN = [2, V2_(1)], ! Set if array in 'column major' order

DBG\$V_PNARR_BITREF = [2, V2_(2)], ! Set if this is a BIT array

DBG\$V_PNARR_RANGE = [2, V2_(3)], ! Set if array has subscript ranges

DBG\$V_PNVAR_VALID = [2, V2_(4)], ! Set if Tag has been validated

DBG\$V_PNODE_IGNORE = [2, V2_(5)], ! Says to ignore SYMID in address computation.

DBG\$P_PNODE_TYPEID = [3, L_], ! TYPEID for this component

DBG\$P_PNODE_SYMID = [4, L_], ! Generic SYMID for this component

DBG\$P_PNODE_RELOC = [5, L_], ! Address relocation factor

DBG\$W_PNREC_INDEX = [6, W0_], ! Record Component Index

DBG\$W_PNREC_NCOMPS = [6, W1_], ! Number of components in record

DBG\$W_PNVAR_INDEX = [6, W0_], ! Variant Component Index

DBG\$W_PNVAR_NCOMPS = [6, W1_], ! Number of components in variant

DBG\$P_PNVAR_TAGID = [7, L_], ! SYMID of tag variable (or 0)

DBG\$P_PNVAR_COMPLST = [8, L_], ! Pointer to list of record components in this record variant

DBG\$P_PNVAR_DSTPTR = [9, L_], ! Pointer to Variant Value DST Record for this record variant

DBG\$B_PNARR_SCALE = [6, B0_], ! Decimal scale factor for array element

DBG\$B_PNARR_DIGITS = [6, B1_], ! Decimal digits for array element or 0

DBG\$B_PNARR_DTYPE = [6, B2_], ! DTYPE of an array element

DBG\$B_PNARR_DIMCNT = [6, B3_], ! Number of array dimensions

DBG\$W_PNARR_LENGTH = [7, W0_], ! Length of each array element

DBG\$B_PNARR_SUBCNT = [7, B3_], ! Number of actual subscripts supplied

DBG\$P_PNARR_OFFSET = [8, L_], ! Offset to element [0,0,...,0] (bit or byte depending on BITREF flag)

DBG\$P_PNARR_CELLTYPE = [9, L_], ! Type ID of array element data type

DBG\$A_PNARR_SVECTOR = [10, A_], ! BLOCKVECTOR of subscripts

TES;

MACRO

DBG\$PRIM_NODE = BLOCK [,LONG] FIELD(DBG\$PNODE_FIELDS)%;

This Primary Descriptor Sub-Node is used for array types (i.e., for Primary Symbol components whose FCODE is Array). The node contains all information needed later about the array type itself, including the dimension count, all subscript types and bounds, etc. This is the format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DBG\$L_PNODE_FLINK			
1	DBG\$L_PNODE_BLINK			
2	Unused	!\$B_PNODE_FLAGS	!\$B_PNODE_FCODE	!\$B_PNODE_KIND
3	DBG\$L_PNODE_TYPEID			
4	DBG\$L_PNODE_SYMID			
5	DBG\$L_PNODE_RELOC			
6	!\$B_PNARR_DIMCNT	!\$B_PNARR_DTYPE	!\$B_PNARR_DIGITS	!\$B_PNARR_SCALE
7	!\$B_PNARR_SUBCNT	Unused	DBG\$W_PNARR_LENGTH	
8	DBG\$L_PNARR_OFFSET			
9	DBG\$L_PNARR_CELLTYPE			
10	DBG\$A_PNARR_SVECTOR			
..	(Block-vector of subscript information)			
..				
..				

The following are the meanings of the fields in the Array Type Sub-Node.

```

DBG$B_PNODE_KIND      -- Must be RST$K_DATA or RST$K_TYPCOMP.
DBG$B_PNODE_FCODE     -- Must be RST$K_TYPE_ARRAY.
DBG$B_PNODE_TYPEID    -- The Type ID for the array type.
DBG$B_PNODE_SYMID     -- The SYMID for the array (or zero).
DBG$B_PNARR_SCALE     -- The element scale factor or zero.
DBG$B_PNARR_DIGITS    -- The element digit count or zero.
DBG$B_PNARR_DTYPE     -- The element VAX standard type code (if
                        applicable) or zero.
DBG$B_PNARR_DIMCNT    -- The number of array dimensions.

```



```

4101 0
4102 0
4103 0
4104 0
4105 0
4106 0
4107 0
4108 0
4109 0
4110 0
4111 0
4112 0
4113 0
4114 0
4115 0
4116 0
4117 0
4118 0
4119 0
4120 0
4121 0
4122 0
4123 0
4124 0
4125 0
4126 0
4127 0
4128 0
4129 0
4130 0
4131 0
4132 0
4133 0
4134 0
4135 0
4136 0
4137 0
4138 0
4139 0
4140 0
4141 0
4142 0
4143 0
4144 0
4145 0
4146 0
4147 0
4148 0
4149 0
4150 0
4151 0
4152 0
4153 0
4154 0
4155 0
4156 0
4157 0

```

```

DBG$B_PNARR_SUBCNT  -- The number of actual subscripts supplied.
                      In some languages this may be less than
                      the number of array dimensions.
DBG$B_PNARR_LENGTH  -- The length of each array element. This
                      length is in bytes except for DTYPES
                      V (bits) and P (packed decimal).
DBG$L_PNARR_OFFSET  -- The offset to element ARR[0,0,...,0]
                      from the start of the array.
DBG$L_PNARR_CELLTYPE -- The Type ID of the array element type.
DBG$A_PNARR_SVECTOR -- Start of the subscript block-vector.

```

The block-vector of subscript information consists of a vector whose elements are blocks of the following format. There is once such block for each dimension in the array.

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	+	-----	+
		DBG\$L_PNSUB_SVALUE	
1	+	-----	+
		DBG\$L_PNSUB_STRIDE	
2	+	-----	+
		DBG\$L_PNSUB_LBOUND	
3	+	-----	+
		DBG\$L_PNSUB_UBOUND	
4	+	-----	+
		DBG\$L_PNSUB_TYPEID	
	+	-----	+

Define the fields in the subscript information block-vector.

FIELD DBG\$PNSUB_FIELDS =

```

SET
DBG$L_PNSUB_SVALUE = [ 0, L_ ], | Subscript Value
DBG$L_PNSUB_STRIDE = [ 1, L_ ], | Subscript Stride (bit or byte
                                | depending on BITREF flag)
DBG$L_PNSUB_LBOUND = [ 2, L_ ], | Subscript Lower Bound
DBG$L_PNSUB_UBOUND = [ 3, L_ ], | Subscript Lower Bound
DBG$L_PNSUB_TYPEID = [ 4, L_ ], | Subscript type's Type ID. If the
                                | subscript type is longword
                                | integer, this field is zero.

```

TES;

MACRO

DBG\$PRIM_NODE_SUBS = BLOCKVECTOR[,5, LONG] FIELD(DBG\$PNSUB_FIELDS) %;

! Define the size of the Primary Descriptor Array Sub-Node.

LITERAL

DBG\$K_PRIM_SIZE_ARRAY = 10, ! Size of fixed part in longwords

: 4158 0
: 4159 0
DBG\$K_PRIM_SIZE_SUBS = 5; ! Size of variable part per subscript,
! also in longwords

PRIMARY DESCRIPTOR RECORD SUB-NODE

This Primary Descriptor Sub-Node is used for record types (i.e., for Primary Symbol components whose FCODE is Record). The only extra field in this sub-node is the record (or "structure") component index which indicates which record component is selected from the defined components of this record type.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DBG\$P_PNODE_FLINK
1	DBG\$P_PNODE_BLINK
2	Unused : \$P_PNODE_FLAGS : \$P_PNODE_FCODE : \$P_PNODE_KIND
3	DBG\$P_PNODE_TYPEID
4	DBG\$P_PNODE_SYMID
5	DBG\$P_PNODE_RELOC
6	DBG\$W_PNREC_NCOMPS : DBG\$W_PNREC_INDEX

The following are the meanings of the fields in the Record Type Sub-Node.

DBG\$B_PNODE_KIND -- Must be RST\$K_DATA or RST\$K_TYPCOMP.
 DBG\$B_PNODE_FCODE -- Must be RST\$K_TYPE_RECORD.
 DBG\$P_PNODE_TYPEID -- The Type ID for the record type.
 DBG\$P_PNODE_SYMID -- The SYMID for the record (or zero).
 DBG\$W_PNREC_INDEX -- The index of the selected record component in the set of record components for this record type. This is meaningful only if the DBG\$V_PNODE_EVAL is set--otherwise this field is always initialized to 1.
 DBG\$W_PNREC_NCOMPS -- The number of components in this record

Define the size of the Primary Descriptor Record Sub-Node.

LITERAL DBG\$K_PRIM_SIZE_RECORD = 7; ! Size of Record Sub-Node in longwords

PRIMARY DESCRIPTOR VARIANT SUB-NODE

This Primary Descriptor Sub-Node is used for variant types (i.e. for Primary Symbol components whose FCODE is variant). The extra fields in this sub-node indicate which is the currently selected component of the variant set (much as a record component is shown in the record sub-node), the number of components, and the SYMID of the associated TAG variable (or zero if no TAG variable) and other information.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DBG\$P_PNODE_FLINK
1	DBG\$P_PNODE_BLINK
2	Unused : \$P_PNODE_FLAGS : \$P_PNODE_FCODE : \$P_PNODE_KIND
3	DBG\$P_PNODE_TYPEID
4	DBG\$P_PNODE_SYMID
5	DBG\$P_PNODE_RELOC
6	DBG\$W_PNVAR_NCOMPS : DBG\$W_PNVAR_INDEX
7	DBG\$P_PNVAR_TAGID
8	DBG\$P_PNVAR_COMPLST
9	DBG\$P_PNVAR_DSTPTR

The following are the meanings of the fields in the VARIANT Type Sub-Node.

DBG\$B_PNODE_KIND -- Must be RST\$K_VARIANT.
DBG\$B_PNODE_FCODE -- Must be RST\$K_TYPE_VARIANT.
DBG\$P_PNODE_TYPEID -- The Type ID for the variant type.
DBG\$P_PNODE_SYMID -- Always Zero.
DBG\$W_PNVAR_INDEX -- The index of the selected component from the set of components for this variant. This field is always initialized to 1.
DBG\$W_PNVAR_NCOMPS -- The number of components in this variant.
DBG\$P_PNVAR_TAGID -- The SYMID of the TAG variable (or zero)
DBG\$P_PNVAR_COMPLST -- Pointer to the list of record components in this record variant.
DBG\$P_PNVAR_DSTPTR -- Pointer to the Variant Value DST Record for this record variant.

G 3
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 98
(55)

```
: 4267 0 ! Define the size of the Primary Descriptor Variant Sub-Node.  
: 4268 0  
: 4269 0 LITERAL  
: 4270 0     DBG$K_PRIM_SIZE_VARIANT = 10; ! Size of Variant Sub-Node in longwords
```


PRIMARY PARSER STATE TABLE

The Primary Parser (DBG\$PRIMARY_PARSER) operates off a Finite-State Machine (FSM) state table which defines the allowed formats of Primary Symbols in the current language. A Primary Symbol consists of a symbol name which may include pathname qualification, data qualification, subscripting, dereferencing, and possibly other qualification depending on the language. In PASCAL, for example, M[R\A.B[2,3]^C[4].D is a valid Primary Symbol. The operators in a Primary Symbol may include '\', '[', '^', and so on. Exactly which operators are allowed and in which order they are allowed is language-dependent, and this is what is specified by the Primary Parser State Table. Each state in the state table is represented by zero or more Primary Parser State Table Transition Entries, each of which specifies a transition to be taken if the current operator is the one specified in the transition entry. The format of a Primary Parser State Table Entry is as follows:

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	First Primary Parser State Table Transition Entry
1	Additional State Table Transition Entries
..	Longword with value zero (0)

The individual Primary Parser State Table Transition Entry has this format:

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	PRIMARY\$W_NEXTSTATE	\$B_ACTION	\$B_OPCODE
---	----------------------	------------	------------

The whole Primary Parser State Table for a language simply consists of a sequence of concatenated State Table Entries, with one entry per state in the Finite-State Machine. The whole state table is thus a vector of transition entries and is indexed as such. Element zero of the state table is the beginning of the state table's Start State, and each transition's next state pointer (PRIMARY\$W_NEXT_STATE) is a state table index which points to the beginning of the next state's state table entry.

A pointer to a Primary Parser State Table is declared as follows:

```

4271 0
4272 0
4273 0
4274 0
4275 0
4276 0
4277 0
4278 0
4279 0
4280 0
4281 0
4282 0
4283 0
4284 0
4285 0
4286 0
4287 0
4288 0
4289 0
4290 0
4291 0
4292 0
4293 0
4294 0
4295 0
4296 0
4297 0
4298 0
4299 0
4300 0
4301 0
4302 0
4303 0
4304 0
4305 0
4306 0
4307 0
4308 0
4309 0
4310 0
4311 0
4312 0
4313 0
4314 0
4315 0
4316 0
4317 0
4318 0
4319 0
4320 0
4321 0
4322 0
4323 0
4324 0
4325 0
4326 0
4327 0

```


PRIMARY_TABLE: REF PRIMARY\$TABLE

This declares PRIMARY TABLE to be a block vector which is referenced as .PRIMARY_TABLE[STATE_INDEX, field-name] where STATE_INDEX is the vector index pointing to the current Primary Parser State Table Transition Entry.

Define the fields in the Primary parser State Table Transition Entry. Also define the declaration macro.

FIELD PRIMARY\$FLD_DEF =

SET
PRIMARY\$B_OPCODE = [0, B0_], | Operator code which activates this
state transition
PRIMARY\$B_ACTION = [0, B1_], | CASE index for semantic action to be
taken during state transition
PRIMARY\$W_NEXTSTATE=[0 , W1_] | Index of next state after transition
TES;

MACRO

PRIMARY\$TABLE = BLOCKVECTOR[,1, LONG] FIELD(PRIMARY\$FLD_DEF) %;

Define the allowed values of the PRIMARY\$B_ACTION field. These values are CASE indices which select the semantic action routine to be executed for each state transition.

LITERAL

PRIMARY\$K_MIN_ACTION	= 1,	--- Minimum action index
PRIMARY\$K_ACT_START_GBL	= 1,	Global scope backslash found
PRIMARY\$K_ACT_GBL_TERM	= 2,	Terminator after global symbol
PRIMARY\$K_ACT_START_SLASH	= 3,	Backslash after first operand
PRIMARY\$K_ACT_START_DOT	= 4,	Dot qualification after first operand
PRIMARY\$K_ACT_START_DOT_PLI	= 5,	" in PL/I
PRIMARY\$K_ACT_START_DOT_COB	= 6,	" in COBOL
PRIMARY\$K_ACT_START_SUBSCR	= 7,	Subscripting after first operand
PRIMARY\$K_ACT_START_SUBSCR_BLI	= 8,	Bliss subscript
PRIMARY\$K_ACT_START_SUBSCR_PLI	= 9,	PL/I subscript
PRIMARY\$K_ACT_START_DEREF	= 10,	PASCAL dereference ^
PRIMARY\$K_ACT_START_DEREF_PLI	= 11,	PL/I "->" operator
PRIMARY\$K_ACT_START_TERM	= 12,	Terminator after first operand
PRIMARY\$K_ACT_START_BIF_CALL	= 13,	Built-in function call
PRIMARY\$K_ACT_START_TICK	= 48,	Ada tick operator
PRIMARY\$K_ACT_SLASH_SLASH	= 14,	Backslash after previous backslash
PRIMARY\$K_ACT_SLASH_INVOCNUM	= 15,	Invocation Number in pathname
PRIMARY\$K_ACT_SLASH_DOT	= 16,	Dot qualification after backslash
PRIMARY\$K_ACT_SLASH_DOT_PLI	= 17,	" in PL/I
PRIMARY\$K_ACT_SLASH_SUBSCR	= 18,	Subscripting after backslash
PRIMARY\$K_ACT_SLASH_SUBSCR_PLI	= 19,	" in PL/I
PRIMARY\$K_ACT_SLASH_SUBSCR_BLI	= 20,	Bliss subscript after backslash
PRIMARY\$K_ACT_SLASH_DEREF	= 21,	PASCAL ^ after backslash
PRIMARY\$K_ACT_SLASH_DEREF_PLI	= 22,	PL/I "->" operator after "/"
PRIMARY\$K_ACT_SLASH_TERM	= 23,	Terminator operator after backslash
PRIMARY\$K_ACT_SLASH_TICK	= 49,	Ada tick operator

4385	0	PRIMARYSK_ACT_DOT_SLASH_COB	= 24.	A of B\C in COBOL
4386	0	PRIMARYSK_ACT_DOT_DOT	= 25.	Dot qualification after previous dot
4387	0	PRIMARYSK_ACT_DOT_DOT_PLI	= 26.	" in PL/I
4388	0	PRIMARYSK_ACT_DOT_DOT_COB	= 27.	" in COBOL
4389	0	PRIMARYSK_ACT_DOT_SUBSCR	= 28.	Subscripting after dot qualification
4390	0	PRIMARYSK_ACT_DOT_SUBSCR_PLI	= 29.	" in PL/I
4391	0	PRIMARYSK_ACT_DOT_SUBSCR_COB	= 30.	" in COBOL
4392	0	PRIMARYSK_ACT_DOT_DEREF	= 31.	PASCAL ^ after dot
4393	0	PRIMARYSK_ACT_DOT_DEREF_PLI	= 32.	PL/I "->" operator after "."
4394	0	PRIMARYSK_ACT_DOT_TERM	= 33.	Terminator operator after dot qualif.
4395	0	PRIMARYSK_ACT_DOT_TERM_PLI	= 34.	" in PL/I
4396	0	PRIMARYSK_ACT_DOT_TERM_COB	= 35.	" in COBOL
4397	0	PRIMARYSK_ACT_DOT_TICK	= 50.	Ada tick operator
4398	0			
4399	0	PRIMARYSK_ACT_SUBSCR_DOT	= 36.	Dot qualification after subscripting
4400	0	PRIMARYSK_ACT_SUBSCR_DOT_PLI	= 37.	" in PL/I
4401	0	PRIMARYSK_ACT_SUBSCR_SUBSCR	= 38.	Subscripting after subscripting
4402	0	PRIMARYSK_ACT_SUBSCR_SUBSCR_PLI	= 39.	" in PL/I
4403	0	PRIMARYSK_ACT_SUBSCR_DEREF	= 40.	PASCAL ^ after []
4404	0	PRIMARYSK_ACT_SUBSCR_DEREF_PLI	= 41.	PLI "->" operator after subscript
4405	0	PRIMARYSK_ACT_SUBSCR_TERM	= 42.	Terminator operator after subscripting
4406	0	PRIMARYSK_ACT_SUBSCR_TERM_PLI	= 43.	" in PL/I
4407	0			
4408	0	PRIMARYSK_ACT_DEREF_DOT	= 44.	Dot after dereferencing
4409	0	PRIMARYSK_ACT_DEREF_SUBSCR	= 45.	Subscripting after dereferencing
4410	0	PRIMARYSK_ACT_DEREF_DEREF	= 46.	Dereferencing after dereferencing
4411	0	PRIMARYSK_ACT_DEREF_TERM	= 47.	Terminator after dereferencing
4412	0			
4413	0	PRIMARYSK_MAX_ACTION	= 50;	--- Maximum action index

PRINT INFORMATION TABLE

The Print Information Table is a blockvector indexed by FCODE which gives pointers to tables and print routine indexes necessary to print primary symbols for a given language.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0  +-----+
  | PRTINFO$ROUT_INDEX |
  +-----+
1  +-----+
  | PRTINFO$CHARTBL    |
  +-----+

```

A pointer to an Print Information Table is declared as follows:

PRINT_INFO_TABLE: REF PRTINFO\$TABLE

Define the fields of the Print Information Table entry. Also define the declaration macro.

```

FIELD PRTINFO$FLD_DEF =
SET
PRTINFO$ROUT_INDEX    = [ 0, L_ ],
PRTINFO$CHARTBL       = [ 1, L_ ],
TES;

```

```

MACRO
PRTINFO$TABLE = BLOCKVECTOR[RST$K_TYPE_MAXIMUM + 1, 2, LONG]
FIELD(PRTINFO$FLD_DEF) %;

```

! Define valid PRTINFO\$ROUT_INDEX literals.

```

LITERAL
PRT$K_MIN_ROUT      = 0,
PRT$K_OTHER         = 0,
PRT$K_ARRAY         = 1,
PRT$K_POINTER       = 2,
PRT$K_RECORD        = 3,
PRT$K_RECORD_COB    = 4,
PRT$K_POINTER_C     = 5,
PRT$K_RECORD_C_PLI  = 6,
PRT$K_VARIANT       = 7,
PRT$K_MAX_ROUT      = 7;

```

! Define valid Print Character Table Indexes.

```

LITERAL
PRT$K_BEGIN_CHAR    = 0,

```

15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 103
(57)

```
: 4471 0      PRISK_SEPARATOR_CHAR = 1.  
: 4472 0      PRISK_END_CHAR       = 2.  
: 4473 0      PRISK_RECPTN_CHAR    = 3.  
: 4474 0      PRISK_MAX_PRTCHAR    = 4;
```


REGISTER DESCRIPTORS

A Register Descriptor is used to describe the absolute address of a register or of a byte address within a register. It thus contains a register number, a byte offset within that register (0, 1, 2, and 3 are the only possible values), and a scope number which indicates to which call frame in the call stack this register belongs. The descriptor also contains a "sentinel" value which is used for validity checking and for ensuring that a Register Descriptor always has a non-zero value.

Register Descriptors are built by routine DBG\$STA_ADDRESS_TO_REGDESCR and are later used for various purposes when registers are examined. They are also required by routine DBG\$STA_REGISTER_NAME which generates the register name for printing.

A Register Descriptor fits in a single longword. This is the format:

```

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+
!  DBG$W_REGD_SCOPENUM  ! $B_REGD_REGNUM ! SENTINEL ! OFF !
+-----+-----+-----+-----+

```

A Register Descriptor is declared as follows:

```

REGDESCR: DBG$REGDESCR

```

Define the fields of the Register Descriptor. Also define the declaration macro.

```

FIELD DBG$REGD_FLD_DEF =

```

```

SET
DBG$V_REGD_OFFSET = [ 0, V_(0,2) ], ! Byte offset from start of register
DBG$V_REGD_SENTINEL = [ 0, V_(2,6) ], ! Sentinel value--must be %X'2D'
DBG$B_REGD_REGNUM = [ 0, B_ ], ! The actual register number
DBG$W_REGD_SCOPENUM = [ 0, W_ ], ! The register's scope number
TES;

```

```

MACRO

```

```

DBG$REGDESCR = BLOCK[1, LONG] FIELD(DBG$REGD_FLD_DEF) %;

```

RUN - TIME SYMBOL TABLE DEFINITIONS

```

: 4523 0
: 4524 0
: 4525 0
: 4526 0
: 4527 0
: 4528 0
: 4529 0
: 4530 0
: 4531 0
: 4532 0
: 4533 0
: 4534 0
: 4535 0
: 4536 0
: 4537 0
: 4538 0
: 4539 0
: 4540 0
: 4541 0
: 4542 0

```

This section contains all definitions related to the structure and contents of the Run-Time Symbol Table (RST) through which the Debugger accesses the Debug Symbol Table (DST) in the user's executable image. The RST itself is built and accessed entirely through the Symbol-Table Access Routines in modules RSTCNTRL and RSTACCESS.

The RST is the structure through which the DST is accessed, one module at a time. This means that for each "active" module in the user's program, there is an RST data structure which describes that module and all lexical entities and data items in it. This structure, which is scattered through the Debugger's free memory, contains one entry for each lexical entity and data item in the module's DST. Each RST entry contains a pointer to the corresponding DST entry so all the DST entry information can be accessed rapidly. In addition, the RST itself is hashed by symbol name so that a symbol's RST and DST entries can be located rapidly given the symbol name.

RST ENTRY COMMON CORE

All entries in the Run-Time Symbol Table have the same information in their first parts. This RST entry "common core" is described here:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT Flag bits RST\$B_KIND

The hash chain forward and backward links link together all RST entries for symbols whose names have the same hash value. The hash chain is doubly linked so that RST entries can be removed from the hash chains when the whole RST for a module is removed to free up its space. Each hash chain has a permanent list head; RST entries in the RST Hash Table therefore never have zero hash links.

Some RST entries are not in the RST Hash Table, however. Such entries are "temporary" in the sense that they are created when needed and are thrown away when no longer referenced. Examples include many Data Type RST Entries and RST entries with invocation numbers. Such RST entries are put on the Temporary RST Entry List. This is a singly linked list which uses the RST\$L_HASH_FLINK field for the links; RST\$L_HASH_BLINK is left zero in this case.

All RST entries for a given module are linked together through the RST\$L_SYMCHNPTR pointer in a singly linked list. The list starts at the Module RST Entry and is terminated by a zero forward link. This list is traversed when the Module's whole RST is released to make room for another module.

The RST\$L_DSTPTR pointer gives the address of the symbol's DST entry. The RST\$L_UPSCOPEPTR pointer generally points to the RST entry of the symbol's "containing" entity. This usually means the containing lexical entity, but for data record components it means the containing record.

The RST\$B_KIND field identifies what kind of symbol and what kind of RST entry this is. The possible values are listed at the end of this section. The RST\$W_REFCOUNT field contains a reference count which specifies how many references (pointers) to this RST entry have been passed

to the rest of the Debugger. When this count is non-zero, this RST entry cannot be released to free storage. The reference count is thus used to prevent a "dangling pointer" problem.

A pointer to a Run-Time Symbol Table Entry is declared as follows:

RSTPTR: REF RST\$ENTRY

This declaration macro collects together all the individual FIELD sets declared below for the various kinds of RST entries.

MACRO

RST\$ENTRY = BLOCK[,LONG] FIELD(RST\$FLD_CORE,RST\$FLD_MOD,RST\$FLD_LEX,RST\$FLD_DATA) %;

These are the RST entry common core field definitions.

FIELD RST\$FLD_CORE =

SET		
RST\$HASH_FLINK	= [0, L_],	Symbol name hash chain forward link
RST\$HASH_BLINK	= [1, L_],	Symbol name hash chain backward link
RST\$SYMCNPNTR	= [2, L_],	Pointer to the next RST entry belong- to the same module as this one
RST\$DSTPTR	= [3, L_],	Pointer to this symbol's DST entry
RST\$UPSCOPEPTR	= [4, L_],	Pointer to the RST entry one level up in scope from this one
RST\$B_KIND	= [5, B0],	The kind of RST entry this is
RST\$V_GLOBAL	= [5, V1_(0)],	Flag set if this is a global symbol
RST\$V_NONZLENGTH	= [5, V1_(1)],	Flag set to TRUE if this symbol has a non-zero length
RST\$V_INVOCNUM	= [5, V1_(2)],	Flag set if symbol has invocation num.
RST\$V_SET_TYPEPTR	= [5, V1_(3)],	Flag set to indicate that the Data RST Entry RST\$TYPEPTR field should be set during RST build.
RST\$V_MARKBIT	= [5, V1_(4)],	Mark bit available for temporary use (such as stopping recursion)
RST\$V_COBOLGBL	= [5, V1_(5)],	Flag bit set to indicate this symbol has the COBOL "global" attribute
RST\$V_REGISTER	= [5, V1_(6)],	Flag set if SYMID represents a register
RST\$W_REFCOUNT	= [5, W1_],	Reference count for storage management
TES;		

The following are the possible values for the RST\$B_KIND field. This field specifies what kind of RST entry this is; it thus indicates both what kind of symbol the RST entry represents and what the format of the RST entry is after the "common core" part.

LITERAL

RST\$K_INVALID	= 0,	Invalid code--cannot occur in RST en- try but can be returned by some Symbol Table Access routines
RST\$K_NOTUNIQUE	= 9,	Symbol not unique--cannot occur in RST entry but can be returned by the


```

: 4657 0
: 4658 0
: 4659 0
: 4660 0
: 4661 0
: 4662 0
: 4663 0
: 4664 0
: 4665 0
: 4666 0
: 4667 0
: 4668 0
: 4669 0
: 4670 0
: 4671 0
: 4672 0

```

RST\$K_MODULE	= 1.	Module RST entry
RST\$K_ROUTINE	= 2.	Routine RST entry
RST\$K_BLOCK	= 3.	Block RST entry
RST\$K_ENTRY	= 8.	Entry Point RST entry
RST\$K_LABEL	= 4.	Label RST entry
RST\$K_LINE	= 5.	Line number RST entry
RST\$K_DATA	= 6.	Data item RST entry
RST\$K_TYPCOMP	= 10.	Type component RST entry
RST\$K_TYPE	= 7.	Data type RST entry
RST\$K_VARIANT	= 11.	Record variant set RST entry
RST\$K_INVOCNUM	= 12.	Invocation number RST Entry
RST\$K_OVERLOAD	= 13.	Overloaded Symbol RST Entry
RST\$K_KIND_MINIMUM	= RST\$K_INVALID,	! Minimum possible kind value
RST\$K_KIND_MAXIMUM	= RST\$K_OVERLOAD;	! Maximum possible kind value

!***-

DBG\$STA_GETSYMBOL routine.

RST ENTRY FOR A MODULE

A module is represented by a Module RST Entry at all times, whether the rest of the module is in the RST or not. This Module Entry then points to a linked list of RST entries for the remaining symbols in the module via the RST\$L_SYMCHNPTR pointer; the end of this list is indicated by a zero pointer. The RST\$L_UPSCOPEPTR pointer is not used as an up-scope pointer in a Module Entry--there is nothing up-scope from a module. Instead RST\$L_NXTMODPTR, which occupies the same location in the RST entry as RST\$L_UPSCOPEPTR, is used to link all Module RST Entries in the whole program into a singly linked list. This list is scanned during RST initialization and when processing the SHOW MODULES command.

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_NXTMODPTR
5	RST\$W_REFCOUNT ; Flag bits ; RST\$B_KIND
6	RST\$L_SAT_PTR
7	RST\$L_PCTBL_BASE
8	RST\$L_MODRSTSIZ
9	RST\$L_MODSRCTBL
10	Unused ; RST\$B_LANGUAGE ; Flag bits
11	RST\$L_MODPCTBL
12	RST\$L_BASEVA
13	Unused ; \$B_IMGFILCHAN

The RST\$L_MODPCTBL field points to a table of pointers to the PC-Correlation Table DST records for this module (used for PC to line number correlation and vice versa). The first cell of this table (namely MOD_PC_TBL[0]) gives the number of PC-Correlation Table DST records in the module's DST, and the remaining cells (i.e., MOD_PC_TBL[1] through MOD_PC_TBL[MOD_PC_TBL[0]]) contain pointers to those DST records. Each cell is one longword. If there are no PC-Correlation Table DST records

4730 0
4731 0
4732 0
4733 0
4734 0
4735 0
4736 0
4737 0
4738 0
4739 0
4740 0
4741 0
4742 0
4743 0
4744 0
4745 0
4746 0
4747 0
4748 0
4749 0
4750 0
4751 0
4752 0
4753 0
4754 0
4755 0
4756 0
4757 0
4758 0
4759 0
4760 0
4761 0
4762 0
4763 0
4764 0
4765 0
4766 0
4767 0
4768 0
4769 0
4770 0
4771 0
4772 0
4773 0
4774 0
4775 0
4776 0
4777 0
4778 0
4779 0
4780 0
4781 0
4782 0
4783 0
4784 0
4785 0
4786 0

for a given module, the RSTSL_MODPCTBL field is zero.

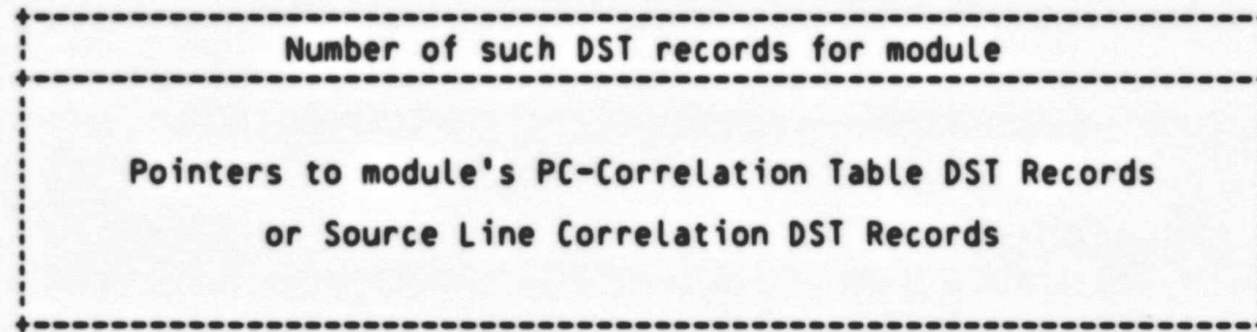
PC-Correlation Table DST records need not have any special order or nesting with respect to the DST records for routines and lexical blocks and are therefore viewed as belonging to the module as a whole. The PC-Correlation Tables for a module are scanned by searching them in the order of their pointers in the RSTSL_MODPCTBL table. The addresses given by the PC Correlation Table DST records are relative to the value of the RSTSL_PCTBL_BASE field, which contains the address of the lowest address routine in the module.

The RSTSL_MODSRCTBL field points to a table of pointers to the Source Line Correlation DST Records for this module. The first cell of this table (i.e., MODSRCTBL[0]) contains the number of Source Line Correlation DST Records in this module's DST and the remaining cells (i.e., MODSRCTBL[1] through MODSRCTBL[MODSRCTBL[0]]) contain pointers to those DST records. Each cell in the table is a longword. If there are no such DST records at all, the RSTSL_MODSRCTBL field is zero.

The memory blocks pointed to by RSTSL_MODPCTBL and RSTSL_MODSRCTBL both have the following format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0
1
..
..
..



When registers are examined using a numbered scope which does not correspond to any routine in the RST, a dummy Module RST Entry is created to represent that scope. In such an RST entry, the RSTSV_MODNUMSCP bit is set to distinguish it from other Module RST Entries. In addition, the scope number (the number of call frames from the top of the call stack) of the desired scope is stored in the RSTSL_MODSCPNUM field. This field overlays the RSTSL_MODRSTSI2 field, which is not needed in this case. The name of such a "module" is just the scope number in decimal ASCII. These special Module RST Entries are built by the GET_REGISTER SYMID routine and are recognized by the DBG\$STA_SETCONTEXT routine. Both in RSTACCESS.

This set of field declarations includes all fields outside the RST common core used in the Module RST Entry.

```

4787 0 FIELD RST$FLD_MOD =
4788 0 SET
4789 0 RST$NXTMODPTR = [ 4, L_ ], | Pointer to the next Module RST Entry
4790 0 RST$SAT_PTR = [ 6, L_ ], | Pointer to this module's Static
4791 0 | Address Table (SAT)
4792 0 RST$PCTBL_BASE= [ 7, L_ ], | The base address for the offsets given
4793 0 | by the PC Correlation Tables
4794 0 RST$MODRSTSIZ = [ 8, L_ ], | The total size in bytes of the RST
4795 0 | for this module--used only by the
4796 0 | SHOW MODULE command.
4797 0 RST$MODSCPNUM = [ 8, L_ ], | The scope number if the MODNUMSCP bit
4798 0 | described below is set
4799 0 RST$MODSRCTBL = [ 9, L_ ], | Pointer to table of pointers to the
4800 0 | module's Source Line Correlation
4801 0 | DST Records (or zero)
4802 0 RST$V_MODSET = [ 10, V_(0) ], | Flag indicating this module is SET
4803 0 RST$V_MOD_IN_RST= [ 10, V_(1) ], | Flag indicating module is in the RST
4804 0 RST$V_ANONMOD = [ 10, V_(2) ], | Flag indicating this is the "anonymous
4805 0 | module" containing all unclaimed
4806 0 | global symbols.
4807 0 RST$V_MODNUMSCP = [ 10, V_(3) ], | Flag indicating that this is a special
4808 0 | RST entry for a numbered scope
4809 0 RST$V_SHARE_IMAGE=[ 10, V_(4) ], | Flag indicating that this is a shared
4810 0 | image
4811 0 RST$V_OLDPLIFLAG= [ 10, V_(5) ], | Flag indicating this is a PL/I module
4812 0 | compiled with an old compiler
4813 0 | which generates incorrect VAX
4814 0 | standard array descriptors
4815 0 RST$B_LANGUAGE = [ 10, B1_ ], | The language of this module
4816 0 RST$MODPCTBL = [ 11, L_ ], | Pointer to a table of pointers to the
4817 0 | module's PC-Correlation Table
4818 0 | DST records (or zero).
4819 0 RST$BASEVA = [ 12, L_ ], | The base address of a shared image
4820 0 RST$B_IMGFILCHAN= [ 13, B_ ] | The executable shared image file
4821 0 | channel number
4822 0 TES;
4823 0
4824 0
4825 0 ! Define a symbolic name for the size of the Module RST Entry in longwords.
4826 0 !
4827 0 LITERAL
4828 0 RST$K_MODENTSIZ = 12; | Size of Module RST Entry
4829 0 RST$K_SHARED_MODENTSIZ = 14; | Size of Shared Image Module RST
4830 0 | Entry

```


The RST entry for a routine has the format shown here. Each Routine RST Entry has an up-scope pointer (RST\$L UPSCOPEPTR) which points to the RST entry for the containing lexical entity. That can be a module, a lexical block, or another routine. The start and end addresses of the routine's code are also given in the RST entry. There is also a Static Address Table (SAT) pointer which points to the SAT entry for this routine, and there is a Static Link pointer which points to the routine's Static Link DST record (or is zero). The routine breakpoint address is also kept in the RST Entry--this address may be set from a Prolog DST Record. The Routine RST Entry is always built when the RST for a module is built.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT ; Flag bits ; RST\$B_KIND
6	RST\$L_STARTADDR
7	RST\$L_ENDADDR
8	RST\$L_RTNSATPTR
9	RST\$L_STATIC_LINK
10	RST\$L_BREAKADDR

This set of field declarations includes the fields outside the RST common core used in the Routine, Lexical Block, Entry Point, Label, and Line Number RST Entries. The individual illustrations show which RST entries use which fields.

FIELD RST\$FLD_LEX =

```

SET
RST$STARTADDR = [ 6, L- ],      ! Lexical entity start address
RST$ENDADDR   = [ 7, L- ],      ! Lexical entity end address
RST$RTNSATPTR = [ 8, L- ],      ! Address of Routine's SAT entry
RST$STATIC LINK = [ 9, L- ],    ! Address of Static Link DST record or 0
RST$BREAKADDR = [ 10, L- ],     ! Routine breakpoint address (start

```

```

: 4888 0
: 4889 0
: 4890 0
: 4891 0
: 4892 0
: 4893 0
: 4894 0
: 4895 0
: 4896 0
: 4897 0
: 4898 0
: 4899 0
: 4900 0
: 4901 0

```

TES;

```

: address, start address + 2, or
: address from Prolog DST Record)

```

```

: Define symbolic names for the lengths of the lexical entity RST entries.
: All lengths are expressed in longwords.

```

LITERAL

```

RST$K_ROUTENTSIZ = 11,      : Size of Routine RST Entry
RST$K_LEXENTSIZ  = 8,      : Size of Lexical Block RST Entry
RST$K_EPTENTSIZ  = 11,     : Size of Entry Point RST Entry
RST$K_LBLENTSIZ  = 7,      : Size of Instruction Label RST Entry
RST$K_LINENTSIZ  = 8;      : Size of Line Number RST Entry

```


RST ENTRY FOR A LEXICAL BLOCK

The RST entry for a lexical block has the same format as that for a routine. Again the up-scope pointer points to the RST entry of the containing lexical entity, and the start and end addresses of the block's code are given. The Lexical Block RST Entry is always built when the RST for the whole module is built.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT ; Flag bits ; RST\$B_KIND
6	RST\$L_STARTADDR
7	RST\$L_ENDADDR

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT Flag bits RST\$B_KIND
6	RST\$L_STARTADDR
7	unused
8	unused
9	unused
10	RST\$L_BREAKADDR

RST ENTRY FOR AN INSTRUCTION LABEL

The RST entry for a label (which labels a point in the user's code) is shown here. The up-scope pointer points to the RST entry of the containing lexical entity. Since a label does not have extent, there is only a start address in the RST entry; no end address is given. The Label RST Entry is always built when the whole module's RST is built.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT Flag bits RST\$B_KIND
6	RST\$L_STARTADDR

RST ENTRY FOR A LINE NUMBER

The RST entry for a line number has the format shown here. The up-scope pointer points to the RST entry of the containing lexical entity (always a routine or a block). Both the start and the end address are given. Line Number RST Entries are not built when the whole module's RST is built, however; there are too many lines in a program to make this practical. Instead each Line Number RST Entry is built when the corresponding line number is referenced through the Symbol Table Access routines. A dummy Label DST Record for the Line Number is built in the same memory block as the Line Number RST Entry. The RST entry points to this DST record and the DST record's name field contains the line number as a counted ASCII string (e.g. "%LINE 25.2").

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT Flag bits RST\$B_KIND
6	RST\$L_STARTADDR
7	RST\$L_ENDADDR

The RST entry for a data symbol has the format shown here. The up-scope pointer points to the RST entry for the containing lexical entity (i.e., module, routine, or lexical block) or, if this is a data component, to the RST entry for the containing data object. The Data Symbol RST Entry also has a Type Pointer (RSTSL_TYPEPTR). For simple data types (those defined by one-byte type codes or standard descriptors) this pointer is zero, but for more complex data types (records, variants, and enumeration types) this pointer points to the RST entry for that data type. If the type pointer is zero, the type can be found directly in the data symbol's DST entry.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK		
1	RST\$L_HASH_BLINK		
2	RST\$L_SYMCHNPTR		
3	RST\$L_DSTPTR		
4	RST\$L_UPSCOPEPTR		
5	RST\$W_REFCOUNT	Flag bits	RST\$B_KIND
6	RST\$L_TYPEPTR		

This set of field declarations includes the fields outside the RST common core used in the Data Symbol, Data Type Component, Data Type, Variant Set, and Invocation Number RST Entries. The individual illustrations show which RST entries use which fields.

FIELD RST\$FLD_DATA =

SET

```
RST$L_TYPEPTR = [ 6, L_ ],
```

```
RST$B_FCODE      = [ 6, B0_ ],
```

```
RSTSW_TYPRFCNT = [ 6, W1_ ],
```

RST\$L_TYPREFTBL = [7, L],

```
RSTSL_BITSIZE = [ 8, 1 ];
```

RST\$L_DST_TYP_REC_PTR =

A pointer to the Data Type RST Entry which gives the data type of this data object. For simple types, this pointer is zero.

The "Format Code" of a Data Type RST Entry--indicates nature of type: array, atomic, record, etc.

The number of Data RST Entries that reference this Type RST Entry

Pointer to Type Reference Table

The length in bits of data items of this data type

A pointer to the DST record containing


```

5094 0          [ 9, L_ ],
5095 0
5096 0
5097 0      RST$L_TYPCOMPNT= [ 10, L_ ],
5098 0
5099 0      RST$A_TYPCOMPLST= [ 11, A_ ],
5100 0
5101 0      RST$L_VARSETCNT = [ 2, L_ ],
5102 0
5103 0      RST$L_VARTAGPTR = [ 4, L_ ],
5104 0
5105 0      RST$A_VARSETTBL = [ 6, A_ ],
5106 0
5107 0
5108 0      RST$L_INVOCNUM = [ 6, L_ ]
5109 0      TES;
5110 0
5111 0
5112 0      ! Declare symbolic names for the lengths of the various data, type, and
5113 0      ! invocation number RST entries. All lengths are in longwords.
5114 0
5115 0      LITERAL
5116 0      RST$K_DATENTSIZ = 7,
5117 0
5118 0      RST$K_OLENTSIZ = 6,
5119 0      RST$K_TYSENTSIZ = 11,
5120 0      RST$K_VARENTSIZ = 6,
5121 0      RST$K_INVENTSIZ = 7;

      the embedded type-spec described
      by the DEBUG built DST record
      pointed to by RST$L_DSTPTR.
      The number of record components if
      this is a data record type
      The start of a table of record compon-
      ent RST pointers
      The number of distinct variants in
      this record variant set
      A pointer to the RST entry of the tag
      variable for this variant set
      The start of a table of variant set
      members, giving the tag value and
      component list for each variant.
      The desired invocation number
  
```

```

      Size of the Data Symbol RST Entry and
      the Type Component RST Entry
      Size of the Overloaded Symbol RST Entry
      Size of the Data Type RST Entry
      Size of the Variant Set RST Entry
      Size of Invocation Number RST Entry
  
```


RST ENTRY FOR A DATA TYPE COMPONENT

The RST entry for a data type component has the format shown here. A "component" in this sense is a record component, one of a set of variants, or one of the enumeration literals of an enumeration type. The interpretation of a data type component is thus dependent on the "type kind" of the containing data type.

The up-scope pointer always points to the RST entry for the data type to which this is a component. Data Type Component RST Entries are always built when the RST for the whole module is built.

Like the Data Item RST Entry, the this RST entry also has a type pointer which specifies the data type of the component. This is zero for simple types or a pointer to a Data Type RST Entry for a more complex data type (such as records).

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT Flag bits RST\$B_KIND
6	RST\$L_TYPEPTR

RST ENTRY FOR A DATA TYPE

The RST entry for a data type has the format shown here. Not all data types have RST entries--those described by one-byte type codes or standard descriptors do not. Data Type RST Entries are used for all record, variant, and enumeration type data types, however. The up-scope pointer always points to the RST entry for the lexical entity (module, routine, or lexical block) in which the data type was declared. A data type need not have a name, however--it can be anonymous as it is in a PL/I or COBOL record declaration. Data Type RST Entries for records, variants, and enumeration types are built when the RST as a whole is built. All other Data Type RST Entries are built only when needed.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT Flag bits RST\$B_KIND
6	RST\$W_TYPREFCNT Unused RST\$B_FCODE
7	RST\$L_TYPREFTBL
8	RST\$L_BITSIZE
9	RST\$L_DST_TYP_REC_PTR
10	RST\$L_TYPCOMPNT
11	RST\$A_TYPCOMPLST
..	Pointers to the Type Component RST Entries of the components of this record or to the Data Item RST Entries of the elements of this enumeration type (Not used for other kinds of data types)

```

5217 0
5218 0
5219 0
5220 0
5221 0
5222 0
5223 0
5224 0
5225 0
5226 0
5227 0
5228 0
5229 0
5230 0
5231 0
5232 0
5233 0
5234 0
5235 0
5236 0
5237 0
5238 0
5239 0
5240 0
5241 0
5242 0
5243 0
5244 0
5245 0

```

These are the possible values for the RST\$B_FCODE field.

LITERAL

RST\$K_TYPE_MINIMUM = 1,	= 1,	--- Minimum possible FCODE value
RST\$K_TYPE_ARRAY	= 2,	Array type
RST\$K_TYPE_ATOMIC	= 3,	Atomic VAX standard type
RST\$K_TYPE_DESCR	= 4,	VAX standard descriptor type
RST\$K_TYPE_ENUM	= 5,	Enumeration type
RST\$K_TYPE_PICT	= 6,	Picture type (as in Cobol and PL/I)
RST\$K_TYPE_TPTR	= 7,	Typed pointer type
RST\$K_TYPE_RECORD	= 8,	Record data type
RST\$K_TYPE_SET	= 9,	Set type
RST\$K_TYPE_SUBRNG	= 10,	Subrange data type
!	= 11,	Unused--available for future use
!	= 12,	Unused--available for future use
RST\$K_TYPE_COBHACK	= 13,	Cobol Hack data item
RST\$K_TYPE_BLIDATA	= 14,	Bliss data item
RST\$K_TYPE_BLIFLD	= 15,	Bliss field
RST\$K_TYPE_FILE	= 16,	File data type (as in Pascal)
RST\$K_TYPE_PTR	= 17,	Untyped pointer data type
RST\$K_TYPE_AREA	= 18,	Area type
RST\$K_TYPE_OFFSET	= 19,	Offset type
RST\$K_TYPE_VARIANT	= 20,	Variant Set (as in Pascal and ADA)
RST\$K_TYPE_RFA	= 21,	Record file address type
RST\$K_TYPE_SELF_REL_LAB	= 22,	Self relative label
RST\$K_TYPE_TASK	= 22,	Task type (as in ADA)
RST\$K_TYPE_MAXIMUM = 22;		--- Maximum possible FCODE value

!***-

RST ENTRY FOR A RECORD VARIANT SET

The RST entry for a record variant set has the format shown here. A record variant set is the set of variants of some data record which are distinguished by the same tag variable. This RST entry does not appear free-standing--it is always built inside the memory block allocated for the containing record's Data Type RST Entry. Its hash links, symbol chain pointer, and up-scope pointers are thus not used. Its DST pointer points to the variant set's Variant-Set Begin DST record. The entry has extra fields which point to the tag variable's RST entry and give the list of individual record variants within the set.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_VARSETCNT
3	RST\$L_DSTPTR
4	RST\$L_VARTAGPTR
5	RST\$W_REFCOUNT ; Flag bits ; RST\$B_KIND
6	RST\$A_VARSETTBL
..	Pointers to the Variant Entries (see next page)
..	for the variants in this variant set
..	(One such pointer per longword)


```

5285 0
5286 0
5287 0
5288 0
5289 0
5290 0
5291 0
5292 0
5293 0
5294 0
5295 0
5296 0
5297 0
5298 0
5299 0
5300 0
5301 0
5302 0
5303 0
5304 0
5305 0
5306 0
5307 0
5308 0
5309 0
5310 0
5311 0
5312 0
5313 0
5314 0
5315 0
5316 0
5317 0
5318 0
5319 0
5320 0

```

This is the format of a Variant Entry as pointed to by the pointers in the Variant-Set RST Entry. This entry gives the list of record components which comprise this particular record variant. It also gives a pointer to the Variant-Value DST record for this variant.

3 3 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RSTSL_VAR_DSTPTR
1	RSTSL_VAR_COMPCNT
2	RSTSA_VAR_COMPLST
..	List of RST pointers to the record components in this variant (One per longword)
..	
..	

Declarations for the fields in the Variant Entry. Also a declaration macro.

FIELD RST\$FLD_VARENT =

SET
RSTSL_VAR_DSTPTR = [0, L_], ! Pointer to Variant-Value DST record
RSTSL_VAR_COMPCNT = [1, L_], ! Number of components in this variant
RSTSA_VAR_COMPLST = [2, A_] ! Start of component RST pointer list
TES;

MACRO

RST\$VAR_ENTRY = BLOCK[,LONG] FIELD(RST\$FLD_VARENT) %; ! Declaration macro

```

5321 0
5322 0
5323 0
5324 0
5325 0
5326 0
5327 0
5328 0
5329 0
5330 0
5331 0
5332 0
5333 0
5334 0
5335 0
5336 0
5337 0
5338 0
5339 0
M 5340 0
5341 0
5342 0

```

These definitions have something to do with variant record tag
 variables. Unfortunately, no comments were put in to say exactly
 what they are used for.

 Define something or another.

FIELD RST\$TAG_BLOCK_FIELDS =
 SET
 RST\$TAG_NUMVALS = [0, L_], ! Number of values to follow (1 or 2)
 RST\$TAG_LOWBOUND = [1, L_], ! Lower bound value
 RST\$TAG_HIGBOUND = [2, L_], ! Upper bound value
 TES;

LITERAL RST\$K_TAG_BLOCK_SIZE = 3; ! The size of the tag variable block

MACRO RST\$TAG_LIST = BLOCKVECTOR[, RST\$K_TAG_BLOCK_SIZE]
 FIELD(RST\$TAG_BLOCK_FIELDS) %;

!***

The RST entry for an invocation number has the format shown here. This RST entry always follows the RST entry to which the invocation number belongs on the module's Symbol Chain. If a Data Item RST Entry, for example, has an associated invocation number, then its RST\$V_INVOCNUM bit is set and its RST\$S_SYMCHNPTR field points to an Invocation Number RST Entry. The Invocation Number RST Entry then specifies the desired invocation number (which is assumed to apply to the innermost routine in the scope of the symbol's declaration). The RST\$S_UPSCOPEPTR field in the Invocation Number RST Entry points to the original RST entry of the object (usually data item) without an invocation number. The format of the Invocation Number RST Entry is as follows:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK		
1	RST\$L_HASH_BLINK		
2	RST\$L_SYMCHNPTR		
3	RST\$L_DSTPTR		
4	RST\$L_UPSCOPEPTR		
5	RST\$W_REFCOUNT	Flag bits	RST\$B_KIND
6	RST\$L_INVOCNUM		

RST ENTRY FOR AN OVERLOADED SYMBOL

The RST entry for an Overloaded Symbol has the format shown below.
 An Overloaded Symbol RST Entry is created for each Overloaded Symbol
 DST record (DST\$K_OVERLOAD). The RST entry is hashed normally so that
 DBG\$STA_GETSYMBOL can find it.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	RST\$L_HASH_FLINK
1	RST\$L_HASH_BLINK
2	RST\$L_SYMCHNPTR
3	RST\$L_DSTPTR
4	RST\$L_UPSCOPEPTR
5	RST\$W_REFCOUNT ; Flag bits ; RST\$B_KIND

SCOPE LIST DEFINITIONS

The scope list maintains the list of scopes declared with the SET SCOPE command. It is a singly linked list pointed to by the global variable SCOPE\$LIST in module RSTCNTRL. Each scope entry on the list has this format:

```

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0	SCOPE\$L_FLINK
1	SCOPE\$L_STATE
2	SCOPE\$L_RSTPTR
3	SCOPE\$L_MODPTR

A pointer to a scope list entry is declared as follows:

```
SCOPEPTR: REF SCOPE$ENTRY
```

Declare the scope entry fields and the definition macro.

```
FIELD SCOPE$FLD_DEF =
```

```

SET
SCOPE$L_FLINK = [ 0, L_ ], ! Scope list forward link or zero
SCOPE$L_STATE = [ 1, L_ ], ! The scope "state" or kind
SCOPE$L_RSTPTR = [ 2, L_ ], ! Pointer to scope's own RST entry
SCOPE$L_MODPTR = [ 3, L_ ], ! Pointer to scope's Module RST Entry
TES;

```

```

LITERAL
SCOPE$K_ENTSIZE = 4; ! Size of scope entry in longwords

```

```

MACRO
SCOPE$ENTRY = BLOCK[SCOPE$K_ENTSIZE] FIELD(SCOPE$FLD_DEF) %;

```

! These are the possible values of the SCOPE\$L_STATE field. Each of these values indicates what kind of scope is to be searched to match a given pathname.

```

LITERAL
SCOPE$K_NORMAL = 1, ! Normal named scope
SCOPE$K_NUMBERED = 2, ! Numbered scope (PC in CALL stack)
SCOPE$K_GLOBAL = 3, ! Global Symbol Table
SCOPE$K_SETMODS = 4; ! All SET modules

```

THE SCREEN DISPLAY ENTRY

The Screen Display Entry contains all information needed to represent and output a screen display on the terminal screen. All Display Entries are linked together by forward and backward links. Each Display Entry contains all information about where the corresponding display is placed on the screen, what attributes it has, and what the contents (the text) of the display is. The text is represented by pointers to the Screen Display Line Entries for the text lines that make up the display.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DBG\$L_DISP_FLINK
1	DBG\$L_DISP_BLINK
2	DBG\$W_DISP_FLAGS : SB_DISP_REND : SB_DISP_KIND
3	Unused : DBG\$W_DISP_SCROLL
4	DBG\$W_DISP_RLEN : DBG\$W_DISP_RBEG
5	DBG\$W_DISP_CLEN : DBG\$W_DISP_CBEG
6	DBG\$W_DISP_DCOL : DBG\$W_DISP_DROW
7	DBG\$W_DISP_LINECNT : DBG\$W_DISP_MAX_LINECNT
8	DBG\$L_DISP_START_LINE_PTR
9	DBG\$L_DISP_END_LINE_PTR
10	DBG\$L_DISP_WINDOW_PTR
11	DBG\$L_DISP_ERROR_PTR
12	DBG\$L_DISP_OLDTXT_PTR
13	DBG\$L_DISP_MODPTR
14	DBG\$L_DISP_CENTER
15	DBG\$L_DISP_MARKLINE
16	DBG\$L_DISP_MINLINE
17	DBG\$L_DISP_MAXLINE
18	DBG\$L_DISP_CMDLIST

19

DBG\$A_DISP_NAME

The display name in Counted ASCII

All Screen Display Entries are linked together on a doubly linked list for which the OWN block DBG\$SCR_DISPLAY_LIST is the list head. Also, the DBG\$L_DISP_START_LINE_PTR and DBG\$L_DISP_END_LINE_PTR fields of the Display Entry constitute the list head for the doubly linked list of Screen Display Line Entries associated with this display.

A pointer to a Display Entry is declared as follows:

DISP_PTR: REF DBG\$DISP_ENTRY

Define the fields of the Screen Display Entry.

FIELD DBG\$DISP_FLD =

SET

DBG\$L_DISP_FLINK	=	[0, L-]	:	Screen display list forward link
DBG\$L_DISP_BLINK	=	[1, L-]	:	Screen display list backward link
DBG\$B_DISP_KIND	=	[2, B0-]	:	Screen display kind
DBG\$B_DISP_REND	=	[2, B1-]	:	Screen display rendition bits
DBG\$V_DISP_REND_BLD	=	[2, V1-(0)]	:	Bolding rendition bit
DBG\$V_DISP_REND_RV	=	[2, V1-(1)]	:	Reverse-video rendition bit
DBG\$V_DISP_REND_BLK	=	[2, V1-(2)]	:	Blinking rendition bit
DBG\$V_DISP_REND_UND	=	[2, V1-(3)]	:	Underline rendition bit
DBG\$W_DISP_FLAGS	=	[2, W1-]	:	Screen display flag bits
DBG\$V_DISP_REMOVE	=	[2, V2-(0)]	:	Remove display from pasteboard
DBG\$V_DISP_INVSCR	=	[2, V2-(1)]	:	Invalidate DISP_SCROLL field
DBG\$V_DISP_ATTOP	=	[2, V2-(2)]	:	Display at top of source file
DBG\$V_DISP_ATBOT	=	[2, V2-(3)]	:	Display at bottom of source file
DBG\$V_DISP_MARKFLG	=	[2, V2-(4)]	:	Mark changed lines in display
DBG\$V_DISP_NEWDISP	=	[2, V2-(5)]	:	New display--suppress marking
DBG\$W_DISP_SCROLL	=	[3, SW0-]	:	Screen display scrolling count
DBG\$W_DISP_RBEG	=	[3, W1-]	:	Unused (available for future use)
DBG\$W_DISP_RLEN	=	[4, W0-]	:	Screen window beginning row location
DBG\$W_DISP_CBEG	=	[4, W1-]	:	Screen window row length (height)
DBG\$W_DISP_CLEN	=	[5, W0-]	:	Screen window beginning column location
DBG\$W_DISP_DROW	=	[5, W1-]	:	Screen window column length (width)
DBG\$W_DISP_DCOL	=	[6, W0-]	:	Screen window display row location
DBG\$W_DISP_MAX_LINECNT	=	[6, W1-]	:	Screen window display column location
DBG\$W_DISP_LINECNT	=	[7, W0-]	:	Screen display's maximum line count
DBG\$W_DISP_LINECNT	=	[7, W1-]	:	(maximum lines saved in memory)
DBG\$L_DISP_START_LINE_PTR	=	[8, L-]	:	Screen display's actual line count
DBG\$L_DISP_START_LINE_PTR	=	[8, L-]	:	Pointer to first line of display text
DBG\$L_DISP_END_LINE_PTR	=	[8, L-]	:	(points to display line entry)
DBG\$L_DISP_END_LINE_PTR	=	[8, L-]	:	Pointer to last line of display text

```

5572 0      DBG$L_DISP_WINDOW_PTR = [ 9, L_ ], (points to display line entry)
5573 0      = [ 10, L_ ], Pointer to first line in screen window
5574 0      DBG$L_DISP_ERROR_PTR = [ 11, L_ ], (points to a Display Line Entry)
5575 0      = [ 12, L_ ], Pointer to list of error message
5576 0      DBG$L_DISP_OLDTXT_PTR = [ 13, L_ ], Display Line Entries for display
5577 0      = [ 14, L_ ], Pointer to list of old Display Line
5578 0      DBG$L_DISP_MODPTR = [ 15, L_ ], Entries--used to mark changes
5579 0      = [ 16, L_ ], Pointer to Module RST Entry--used for
5580 0      DBG$L_DISP_CENTER = [ 17, L_ ], source line displays only
5581 0      DBG$L_DISP_MARKLINE = [ 18, L_ ], Central line number for source display
5582 0      = [ 19, L_ ], Line number to be marked with "->" in
5583 0      DBG$L_DISP_MINLINE = [ 20, L_ ], source line display (or -1)
5584 0      DBG$L_DISP_MAXLINE = [ 21, L_ ], Minimum source line number in module
5585 0      DBG$L_DISP_CMDLIST = [ 22, L_ ], Maximum source line number in module
5586 0      = [ 23, L_ ], Pointer to display's DEBUG command
5587 0      list entry (or zero)
5588 0      DBG$A_DISP_NAME = [ 24, A_ ] The display's name in Counted ASCII
5589 0      TES;

```

MACRO

```
DBG$DISP_ENTRY = BLOCK[,LONG] FIELD(DBG$DISP_FLD) %;
```

LITERAL

```
DBG$K_DISP_ENTSIZE = 19; ! Size of the fixed part of the Screen
                          ! Display Entry in longwords
```

```
! Define the literals used to indicate the display kind. The display kind
! determined how the contents of the display are generated.
```

LITERAL

```

DBG$K_DISP_MINKIND = 0, ---Minimum kind value
DBG$K_DISP_NOKIND = 0, No kind--used in calls to indicate no
                        change to the display's kind
DBG$K_DISP_NORMAL = 1, Normal display kind--contents deter-
                        mined by direct user input
DBG$K_DISP_DO = 2, Contents specified by DO command list
DBG$K_DISP_SOURCE = 3, Source display with contents speci-
                        fied by a source command list
DBG$K_DISP_REGISTER = 4, Contents is machine register display
DBG$K_DISP_MAXKIND = 4; ---Maximum kind value

```

```
! Define literals for the display rendition bits.
```

LITERAL

```

DBG$M_DISP_REND_BLD = 1, ! Bolding rendition mask
DBG$M_DISP_REND_RV = 2, ! Reverse-video rendition mask
DBG$M_DISP_REND_BLK = 4, ! Blinking rendition mask
DBG$M_DISP_REND_UND = 8, ! Underline rendition mask

```

```
! Define the literals used to indicate the scrolling direction when scrolling
! a screen display.
```

LITERAL

```
DBG$K_SCROLL_UP = 1, ! Scroll the display up
```


8 6
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 132
(77)

```
: 5629 0      DBG$K_SCROLL_DOWN      = 2;  ! Scroll the display down
: 5630 0      DBG$K_SCROLL_LEFT      = 3;  ! Scroll the display to the left
: 5631 0      DBG$K_SCROLL_RIGHT     = 4;  ! Scroll the display to the right
```

5632	0
5633	0
5634	0
5635	0
5636	0
5637	0
5638	0
5639	0
5640	0
5641	0
5642	0
5643	0
5644	0
5645	0
5646	0
5647	0
5648	0
5649	0
5650	0
5651	0
5652	0
5653	0
5654	0
5655	0
5656	0
5657	0
5658	0
5659	0
5660	0
5661	0
5662	0
5663	0
5664	0
5665	0
5666	0
5667	0
5668	0
5669	0
5670	0
5671	0
5672	0
5673	0
5674	0
5675	0
5676	0
5677	0
5678	0

Screen Display Line Entries come in two variants, the Normal and the Source Display Line Entry. Source Display Line Entries are used for source displays as the name implies, and Normal Display Line Entries are used for all other kinds of displays. Normal Display Line Entries can optionally include rendition information on a per-character basis.

The Normal Display Line Entry is used for all screen displays except Source screen displays. This is the format of the Normal Screen Display Line Entry:

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

```

0 |-----+-----+-----+-----+
  |DBG$ _DLIN E_ FLINK
1 |-----+-----+-----+-----+
  |DBG$ _DLIN E_ BLINK
2 |-----+-----+-----+-----+
  |Unused   ! $B _DLIN E_ LENGTH!    Flags      ! $B _DLIN E_ REND
  |-----+-----+-----+-----+
3 |               DBG$A _DLIN E_ TEXT
..
.. |               The Counted ASCII text of the display line
..
.. |

```


NORMAL DISPLAY LINE ENTRY WITH RENDITION VECTOR

The Normal Display Line Entry may include a "rendition vector" if the screen rendition is not the same for the entire line. In this case, the DBG\$V_DLINE_RENDFLG is set in the Display Line Entry's flag field and a "rendition vector" is present at the end of the Display Line Entry, immediately following the Counted ASCII text of the display line. The rendition vector contains one byte of rendition codes for each character in the Counted ASCII text of the line. The rendition vector has the same length as the Counted ASCII text.

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DBG\$L_DLINE_FLINK
1	DBG\$L_DLINE_BLINK
2	Unused !SB_DLINE_LENGTH! Flags !SB_DLINE_REND
3	DBG\$A_DLINE_TEXT
..	The Counted ASCII text of the display line
..	
..	
..	
..	The Rendition Vector (one byte per text character)
..	
..	

5679 0
5680 0
5681 0
5682 0
5683 0
5684 0
5685 0
5686 0
5687 0
5688 0
5689 0
5690 0
5691 0
5692 0
5693 0
5694 0
5695 0
5696 0
5697 0
5698 0
5699 0
5700 0
5701 0
5702 0
5703 0
5704 0
5705 0
5706 0
5707 0
5708 0
5709 0
5710 0
5711 0
5712 0
5713 0
5714 0
5715 0
5716 0
5717 0

FIELD DBG\$DLN_FLD =


```

5775 0
5776 0
5777 0
5778 0
5779 0
5780 0
5781 0
5782 0
5783 0
5784 0
5785 0
5786 0
5787 0
5788 0
5789 0
5790 0
5791 0
5792 0
5793 0
5794 0
5795 0
5796 0
5797 0
5798 0
5799 0
5800 0
5801 0
5802 0
5803 0

```

SET

DBG\$DLINE_FLINK = [0, L_], ! Forward link to next line entry

DBG\$DLINE_BLINK = [1, L_], ! Backward link to previous line entry

DBG\$DLINE_REND = [2, B0], ! Rendition bits for this line

DBG\$DLINE_REND_BLD = [2, V0-(0)], ! Bolding rendition bit

DBG\$DLINE_REND_RV = [2, V0-(1)], ! Reverse-video rendition bit

DBG\$DLINE_REND_BLK = [2, V0-(2)], ! Blinking rendition bit

DBG\$DLINE_REND_UND = [2, V0-(3)], ! Underline rendition bit

! = [2, B1], ! The flag byte

DBG\$DLINE_RENDFLG = [2, V1-(0)], ! Line includes rendition vector

DBG\$DLINE_SOURCEFLG = [2, V1-(1)], ! Set for Source Disp Line Entry

DBG\$DLINE_LENGTH = [2, B2], ! The length of the text buffer (bytes)

DBG\$DLINE_TEXT = [3, A_], ! The line's text in Counted ASCII

DBG\$DLINE_FILEID = [2, W1], ! Source file ID of source line

DBG\$DLINE_RECNUM = [3, L_], ! Source file record number of line

DBG\$DLINE_LINUM = [4, L_], ! Line number of source line

DBG\$DLINE_TEXT2 = [5, A_], ! Source line's text in Counted ASCII

TES;

MACRO

DBG\$DLINE_ENTRY = BLOCK[,LONG] FIELD(DBG\$DLINE_FLD) %;

LITERAL

DBG\$K_DLINE_ENTSIZE = 3, ! Size of fixed part of Normal Display

! Line Entry in longwords

DBG\$K_DLINE_ENTSIZE2 = 5; ! Size of fixed part of Source Display

! Line entry in longwords

THE SCREEN PASTEBOARD ENTRY

The screen pasteboard, on which all displays to be output on the terminal are pasted to determine what occludes what, is represented by a blockvector. The vector is indexed by line number and has one entry (block) for each line on the screen. The length of the vector is actually 21 (not 24) so that the last three lines on the screen are always reserved for user program output and DEBUG input prompting. For each line, the vector contains one Pasteboard Entry of the following format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	DBGSW_PASTE_SCROLL	SB_PASTE_REND	SB_PASTE_KIND
1	DBGSL_PASTE_DISPID		
2	DBGSL_PASTE_DLINE		

Define the fields of the individual Pasteboard Entry. Also declare the declaration macro and the size of the vector and of each block.

```
FIELD DBG$PASTE_FLD =
SET
DBG$B_PASTE_KIND = [ 0, B0 ], ! The kind of line this entry holds
DBG$B_PASTE_REND = [ 0, B1 ], ! The rendition bits for this line
DBG$W_PASTE_SCROLL = [ 0, SW ], ! The scrolling amount for this line
DBG$L_PASTE_DISPID = [ 1, L ], ! Pointer to Display Entry for line
DBG$L_PASTE_DLINE = [ 2, L ], ! Pointer to Display Line Entry
TES;
```

```
MACRO
DBG$PASTEBOARD =
BLOCKVECTOR[DBG$K_PASTE_SIZE, DBG$K_PASTE_ENTSIZE, LONG]
FIELD(DBG$PASTE_FLD) %;
```

```
LITERAL
DBG$K_PASTE_SIZE = 21, ! Number of lines in pasteboard
DBG$K_PASTE_ENTSIZE = 3; ! Size of one Pasteboard Entry in
! longwords
```

! Define the allowed values of the DBG\$B_PASTE_KIND field.

```
LITERAL
DBG$K_PASTE_NULL = 1, ! Null line--no display covers it
DBG$K_PASTE_TEXT = 2, ! Text line from a display
DBG$K_PASTE_BLANK = 3, ! Blank line within a display
DBG$K_PASTE_LABEL = 4, ! Top border line with label
```


H 6
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 138
(81)

; 5861 0 DBG\$K_PASTE_BORDER = 5; ! Bottom border line on screen

[illegible]

J 6
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 140
(83)

SOURCE DIRECTORY SEARCH LIST

The Source Directory Search List is a list structure which keeps track of all source directory names specified by the user via SET SOURCE and SET SOURCE/MODULE=xxx commands. This structure is thus searched when a source file must be opened so that the file is opened in the proper directory.

The structure of this list is as follows. Variable DBFSSRC DIR_LIST in module DBGSOURCE points to a singly linked list of Source Directory Search List Header Blocks. Each Header Block contains a Module RST Entry pointer (which may be zero) and points to a singly linked list of Source Directory Search List Entries. Each such entry contains a directory name as a Counted ASCII string.

The list is searched by first searching all the Header Blocks for the Module RST Entry pointer for the current module, i.e. the module from which source lines are to be displayed. If the desired module is not found, the Header Block with the zero Module RST Entry pointer is used instead. The found Header Block then points to the linked list of directory names to use when searching for the desired source file. If there is no Header Block with either the desired module pointer or the zero module pointer, no Source Directory Search List Entries are used--the file name from the Declare Source File DST command is used as is.

The command SET SOURCE/MODULE=modname dir1,...,dirN is represented by a Header Block with a pointer to the Module RST Entry for module "modname" and a pointer to a linked list of Entries. There is one Entry for each of dir1, ..., dirN. The command SET SOURCE dir1,...,dirN is represented by a Header Block with a zero Module RST Entry pointer and of course a pointer to a linked list of Entries for dir1, ..., dirN.

SOURCE DIRECTORY SEARCH LIST HEADER BLOCK

The Source Directory Search List Header Block, as described on the previous page, has the following format:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	SDSL\$FLINK
1	SDSL\$LIST_PTR
2	SDSL\$MODPTR

A pointer to a Source Directory Search List Header Block is declared as follows:

SDSL_PTR: REF SDSL\$HEADER

Declare the fields of the Source Directory Search List Header Block. Also declare the declaration macro.

FIELD SDSL\$FLD_HEADER =

SET
SDSL\$FLINK = [0, L_], ! Forward link to next Header Block
SDSL\$LIST_PTR = [1, L_], ! Pointer to list of SDSL Entries
SDSL\$MODPTR = [2, L_], ! Pointer to Module RST Entry of module
! to which search list applies; is
! zero if for all other modules

TES;

LITERAL

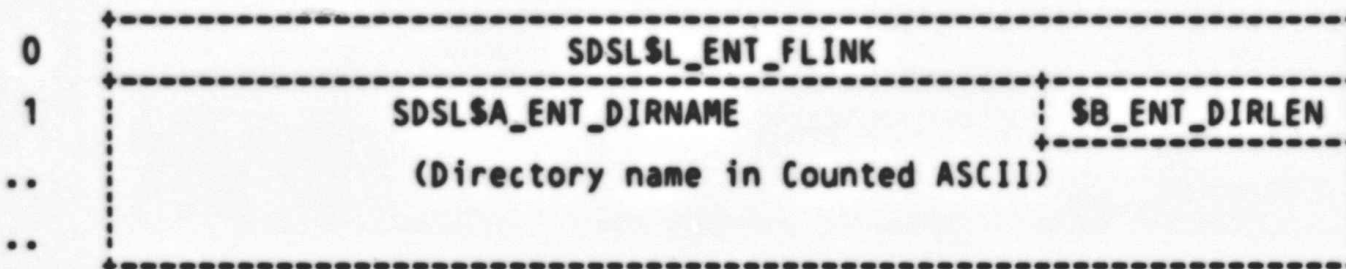
SDSL\$K_HDR_SIZE = 3; ! Size of SDSL Header Block in longwords

MACRO SDSL\$HEADER = BLOCK[SDSL\$K_HDR_SIZE] FIELD(SDSL\$FLD_HEADER) %;

SOURCE DIRECTORY SEARCH LIST ENTRY

There is one Source Directory Search List Entry for each directory name specified on a SET SOURCE command. The "directory name" may in fact contain a full file name, including extension and version number, but in most cases it is used only to override the directory name as given in the Declare Source File DST command. This is the format of a Source Directory Search List Entry:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



A pointer to a Source Directory Search List Entry is declared like this:

SDSL_ENT_PTR: REF SDSL\$ENTRY

Declare the fields of the Source Directory Search List Entry. Also declare the declaration macro.

FIELD SDSL\$FLD_ENTRY =

SET
SDSL\$ENT_FLINK = [0, L], ! Forward link to next SDSL Entry
SDSL\$B_ENT_DIRLEN = [1, B0], ! Length in characters of directory name
SDSL\$A_ENT_DIRNAME = [1, A1], ! First character of directory name
TES;

LITERAL

SDSL\$K_ENT_SIZE = 1; ! Size of fixed portion of SDSL Entry
! in longwords

MACRO SDSL\$ENTRY = BLOCK[,LONG] FIELD(SDSL\$FLD_ENTRY) %;

SOURCE FILE CONTROL BLOCK

This is the format of the Source File Control Block:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	SFCBSL_FLINK
1	SFCBSL_BLINK
2	Unused (MBZ) SFCBSB_KIND
3	SFCBSW_RFACURLEN SFCBSW_RFASPACING
4	SFCBSL_RFATBLPTR
5	SFCBSL_DSTPTR
6	SFCBSL_FABPTR
7	SFCBSL_RABPTR
8	SFCBSL_NAMPTR
9	SFCBSL_XABDATPTR
10	SFCBSL_XABFHCPTR
11	SFCBSL_NAMBUFFER
12	SFCBSL_CURRECNUM
13	SFCBSL_CUR_RFA0
14	Unused SFCBSW_CUR_RFA4
15	SFCBSL_LBRINDEX

The Source File Control Blocks (SFCBs) keep track of all currently open source files. This includes modules within source libraries as well as normal RMS files. Each such block contains all information needed to read source records from the corresponding file. These blocks form a circular doubly linked list, where variable DBGSSRC_SFCB_PTR in module DBGSOURCE points to the first block on the list. The blocks are always maintained in order so that the Most Recently Used SFCB is first on the list and the Least Recently Used block is last. This ordering allows DEBUG to close the Least Recently Used source file when a file must be closed in order to open a new file.

DEBUG always keeps a fixed number of Source File Control Blocks on the list. This number is given by variable DBG\$SRC_MAX_FILES in module DBG\$SOURCE. It can be changed by the user with the SET MAX_SOURCE_FILES command. This number limits the number of source files which are open at the same time. Such a limit is required to prevent DEBUG from using up too many of the limited number of channels the user can have open at any one time. Thus, when all SFCBs are in use and a new source file must be opened, the Least Recently Used file is closed and its SFCB is reused for the new file.

A pointer to a Source File Control Block is declared as follows:

SFCB_PTR: REF SFCB\$BLOCK

Declare the fields of the Source File Control Block. Also declare the declaration macro.

FIELD SFCB\$FLD_DEF =

SET		
SFCB\$L_FLINK	= [0, L_],	! Forward link to next SFCB
SFCB\$L_BLINK	= [1, L_],	! Backward link to previous SFCB
SFCB\$B_KIND	= [2, B0_],	! The kind of source file this is
SFCB\$W_RFASPCING	= [3, W0_],	! The allocated length of the RFA table
SFCB\$W_RFACURLEN	= [3, W1_],	! The used length of the RFA table
SFCB\$L_RFATBLPTR	= [4, L_],	! Pointer to Record File Address (RFA) table for this source file
SFCB\$L_DSTPTR	= [5, L_],	! Pointer to DST Declare Source File command for this source file
SFCB\$L_FABPTR	= [6, L_],	! Pointer to the FAB for this file
SFCB\$L_RABPTR	= [7, L_],	! Pointer to the RAB for this file
SFCB\$L_NAMPTR	= [8, L_],	! Pointer to the NAM block for this file
SFCB\$L_XABDATPTR	= [9, L_],	! Pointer to the XAB Date and Time block for this file
SFCB\$L_XABFHCPTR	= [10, L_],	! Pointer to XAB File Header Characteristics block for this file
SFCB\$L_NAMBUFFER	= [11, L_],	! Pointer to buffer for NAM block file name strings for this file
SFCB\$L_CURRECNUM	= [12, L_],	! Record number at which the source file is currently positioned
SFCB\$L_CUR_RFA0	= [13, L_],	! RFA at which the source file is currently positioned (first 4 bytes)
SFCB\$W_CUR_RFA4	= [14, W0_],	! RFA at which the source file is currently positioned (last 2 bytes)
SFCB\$L_LBRINDEX	= [15, L_]	! Library file index used by librarian

TES;

LITERAL SFCB\$K_SIZE = 16; ! Size of SFCB in longwords

MACRO SFCB\$BLOCK = BLOCK[SFCB\$K_SIZE] FIELD(SFCB\$FLD_DEF) %;

! Declare the possible values of the SFCB\$B_KIND field.

LITERAL SFCB\$K_NOTUSED = 1, ! This SFCB is not used (is available)

: 6156 0
: 6157 0
: 6158 0
: 6159 0
: 6160 0

SFCB\$K_RMSFILE = 2,
SFCB\$K_LBRFILE = 3,
SFCB\$K_FILE_UNAVAIL = 4;

! This SFCB is used for an RMS file
! This SFCB is used for a module within
! a source library
! This SFCB is occupied by a file which
! is presently not available

SOURCE FILE ID TABLE

The Source File ID Table is used during the decoding of the Source Line Correlation DST Record to keep track of the File IDs used to reference the current module's source files. The Source File ID Table consists of a singly linked list of Source File ID Table Entries, where each entry gives a File ID, a pointer to the corresponding Declare Source File command, (This command gives all needed information about the identity and nature of the source file.), and the current source record number. This table is used locally in routine DBG\$SRC_TYPE_LNUM_SOURCE in module DBG\$SOURCE. This is the format of each Source File ID Table Entry:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

0	----- SFIT\$L_FLINK -----
1	----- SFIT\$L_FILE_ID -----
2	----- SFIT\$L_DSTPTR -----
3	----- SFIT\$L_CURRECNUM -----

A pointer to a Source File ID Table Entry is declared as follows:

SFIT_PTR: REF SFIT\$ENTRY

Declare the fields of the Source File ID Table Entry. Also declare the declaration macro.

FIELD SFIT\$FLD_DEF =

SET
 SFIT\$L_FLINK = [0, L_], ! Forward link to next SFIT entry
 SFIT\$L_FILE_ID = [1, L_], ! File ID value for this source file
 SFIT\$L_DSTPTR = [2, L_], ! Pointer to DST Declare Source File
 ! command for this source file
 SFIT\$L_CURRECNUM = [3, L_] ! Current source record number for
 ! this source file
 TES;

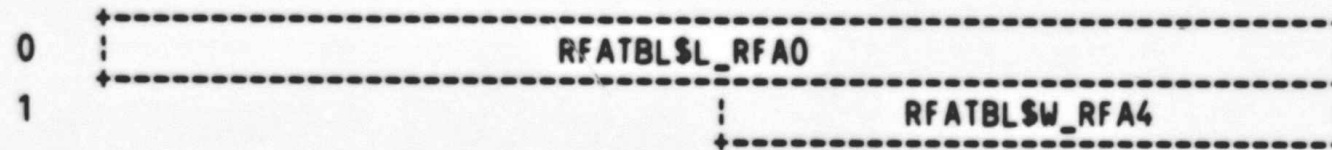
LITERAL SFIT\$K_SIZE = 4; ! Size of SFIT entry in longwords

MACRO SFIT\$ENTRY = BLOCK[SFIT\$K_SIZE] FIELD(SFIT\$FLD_DEF) %;

SOURCE FILE RECORD FILE ADDRESS TABLE

This is the format of each Record File Address Table entry:

3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0



The Record File Address Table correlates record numbers in a source file with RFA's for that file. It is used during the TYPE command and other commands that cause source lines to be displayed, to help locate the desired record more efficiently.

There is one RFA table for each open file. The table is allocated during DBG\$SRC_INIT and a pointer to the table is placed in the Source File Control Block. The table consists of a fixed number of entries (RFATBL_SIZE_ENTRIES, defined in DBG\$SOURCE).

For every N records in the file, there is one RFA entry. That is, the first entry is the RFA for record 1, the next for record (1+N), the next for record (1+2N), and so on. The spacing N can be adjusted dynamically and is kept in the Source File Control Block.

A pointer to an RFA table is declared as follows:

```
RFATBL_PTR : REF RFATBL$BLOCKVECTOR(RFATBL_SIZE_ENTRIES)
```

Declare the fields of an RFA table entry. Also declare the declaration macro.

FIELD RFATBL\$FLD_DEF =

```

  SET
  RFATBL$RFA0 = [ 0 , L_ ] , ! Block number part of RFA
  RFATBL$W_RFA4 = [ 4 , W0_ ] , ! Byte offset part of RFA
  TES;
  
```

LITERAL

```
RFATBL$K_SIZE = 6; ! Table entry size in bytes
```

MACRO

```

RFATBL$BLOCKVECTOR(N) =
  BLOCKVECTOR [N, RFATBL$K_SIZE, BYTE] FIELD(RFATBL$FLD_DEF) %;
  
```


STATIC ADDRESS TABLE DEFINITIONS

The Static Address Table (SAT) consists of a two-level structure. There is a Program Static Address Table which specifies what static addresses (code addresses and static data addresses) belong to what modules. This table consists of a linked list of SAT entries where each entry specifies an address range and a pointer to the corresponding Module RST Entry. The list is ordered by start address so that low start addresses come before higher start addresses. It should be noted that a given address may be covered by more than one SAT entry because different modules can have global PSECTs in common--Fortran COMMON blocks cause this situation to arise. Given an address, the Program Static Address Table can thus be searched to find the module that contains the address.

On a second level, there is a Module Static Address Table for each module that is SET. This SAT is a linked list of exactly the same structure as the Program SAT except that the RST pointer in the SAT entry points to the RST entry of the symbol, not the module, containing the address range. This list is also ordered by start address and again more than one symbol may contain the same address (due to Fortran EQUIVALENCEing for example). If two entries have the same start address, the entry with the larger end address is placed first in the Module SAT; this is crucial to the correct behavior of the DBG\$STA_SETCONTEXT routine where routine addresses are compared to PC values from the VAX call stack. The Module RST Entry contains a pointer (RST\$SAT_PTR) which points to the first SAT entry on the module's SAT chain.

Note (R. Title)- for a future release of DEBUG, I intend to turn the SAT chains into binary trees, instead of linked lists. The reason for this is that for large programs, the SAT chains can grow quite long (e.g., 1000 entries), and it becomes quite expensive to search this list linearly.

The tree search algorithm will work something like this: If the desired address is greater than the address in the SAT\$END field of the current node, follow the "right son" pointer. If it is less than the address in the SAT\$START field of the current node, then follow the "left son" pointer. If it is in the range, then we have found the desired SAT entry and we are done.

During the transition to this new data structure, the same four fields are being kept in the SAT entry in the same place (so that the old code still works). The "right son" field is overlaid with the FLINK field, so that the tree search algorithm will work on the old linked lists (a linear list is just an extremely unbalanced tree, with no left branches). The "left son" field will occupy the fifth longword.

The individual Static Address Table entry has this format:

3 3 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

----->

Address	Value	Field
6323	0	SATSL_FLINK (also SATSL_RIGHTSON)
6324	0	
6325	1	SATSL_START
6326	0	
6327	2	SATSL_END
6328	0	
6329	3	SATSL_RSTPTR
6330	0	
6331	4	SATSL_LEFTSON
6332	0	

Value Descriptors describe language values generated during the processing of language expressions in DEBUG commands like EVAL and DEPOSIT. The language-independent version of the Value Descriptor consists of the Value and Primary Descriptor header fields followed by a 3-longword VAX Standard Descriptor followed by the actual value described by the Value Descriptor. It thus has the following format:

```

SET
DBG$W_VALUE_TOKENCODE=[ 4, WO_ ],      ! Value of TOKEN$W_CODE for constants

```

```

6417 0      DBG$W_VALUE_SIGN_CODE=[ 4, W1_ ],      ! Sign code (Value of Unary -/+
6418 0      ! TOKENSW_CODE) for constants
6419 0      DBG$A_VALUE_VMSDESC = [ 5, A_ ],      ! Address of VAX/VMS descriptor
6420 0      DSC$B_CLASS sub-field
6421 0      DSC$B_DTYPE sub-field
6422 0      DSC$W_LENGTH sub-field
6423 0      DSC$A_POINTER sub-field
6424 0      DSC$B_SCALE sub-field
6425 0      DSC$B_DIGITS sub-field
6426 0      DSC$V_FL_BINSCALE = [7, 19, 1, 0], ! DSC$V_FL_BINSCALE sub-field
6427 0      DSC$L_POS sub-field
6428 0      DSC$A_ADDRESS = [ 8, A_ ],      ! First byte of value
6429 0      DSC$L_VALUE0 = [ 8, L_ ],      ! First longword of value
6430 0      DSC$L_VALUE1 = [ 9, L_ ],      ! Second longword of value
6431 0      TES;
6432 0
6433 0      MACRO
6434 0      DBG$VALDESC = BLOCK [,LONG] FIELD(DBG$DHDR_FIELDS, DBG$VALUE_FIELDS) %;
6435 0
6436 0      ! Define the "base size" of a value descriptor in longwords. For the cases
6437 0      ! where the value is actually stored in the descriptor, the total size
6438 0      ! is the base size plus the number of longwords needed to hold the actual
6439 0      ! value.
6440 0
6441 0      LITERAL
6442 0      DBG$K_VALDESC_BASE_SIZE = 8;
6443 0

```


OLD DEBUGGER DEFINITIONS

These definitions are used by old Debugger modules which also need to use the new RST (and other) definitions.

LITERAL

! constants used to identify individual rst flags
rst\$f_global = 1,
rst\$f_nonzlength = 2,
rst\$f_modset = 3;

MACRO

! Get the srm type address from a dst with address calculation commands
! The type is following the name string
! 7 - size of fixed portion of dst
! 1 - byte to hold count of name string
! dst[dst_name] - contains count of characters in name string
! add this distance to base address to get address of byte
! containing the srm type

CAD_SRM_TYPEA(DST) =
(.DST + 7 + 1 + .DST[dst\$b_name]) %,

! Get the srm type from a dst with address calculation commands

CAD_SRM_TYPE(DST) =
(.(CAD_SRM_TYPEA(DST))<0,8,0>) %,

! Get the address of the calculation commands in a dst which contains
! them. They start one byte past the srm type.

CAD_COMMANDS(DST) =
(CAD_SRM_TYPEA(DST) + 1) %;

MACRO

RST_UNITS(bytes) =
(((bytes) + %upval-1)/%upval)
%;

MACRO

! DEBUG tells the RST module about ASCII
! strings by passing a counted string pointer.
CS_POINTER = REF VECTOR[1,BYTE] %;

LITERAL

! We will never print "symbol+offset" when the
! upper bound for "symbol" is 0 and when
! the offset is greater than RST_MAX_OFFSET

7
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 153
(91)

6501 0
6502 0
6503 0
6504 0
6505 0
6506 0
6507 0
6508 0
6509 0
6510 0
6511 0
6512 0
6513 0

RST\$K_MAX_OFFSET = %x'100';

!+
Since scope definitions are recursive, we must
stack ROUTINE BEGINS in the routine ADD_MODULE.
It is no coincidence that this stack limit is the
same as the limit on the length (in elements) of
symbol pathnames.
!-

LITERAL
MAX_SCOPE_DEPTH = dbg\$K_max_pathname; ! Routines can be nested to a maximum depth.

K 7
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255SDUA28:[DEBUG.SRC]DBGLIB.REQ;1

Page 154
(92)

```
6514 0
6515 0
6516 0
6517 0
6518 0
6519 0
6520 0
6521 0
6522 0
6523 0
6524 0
6525 0
6526 0
6527 0
6528 0
6529 0
6530 0
6531 0

FIELD VALU_FIELD_SET =
SET VALU_NT_PTR = [ 0,0,32,0 ], ! Associated NT pointer.
VALU_VALUE = [ 4,0,32,0 ] ! The actual value.
TES;

!+
! Declare an occurrence or REF to a VALUE_DESCRIPTOR
! via the following macros.
!-

LITERAL VALU_DESC_SIZE = 8; ! Each one is 2 longwords long.

MACRO VALU_DESCRIPTOR = BLOCK[ VALU_DESC_SIZE, BYTE ] FIELD( VALU_FIELD_SET ) %;
```

```
6532 0
6533 0
6534 0
6535 0
6536 0
6537 0
6538 0
6539 0
6540 0
6541 0
6542 0
6543 0
6544 0
6545 0
6546 0
6547 0
6548 0
6549 0
6550 0
6551 0
6552 0
6553 0
6554 0
6555 0
6556 0
6557 0
6558 0
6559 0
6560 0
6561 0
6562 0
6563 0
6564 0
6565 0
6566 0
6567 0
6568 0
```

```
++
Array Bounds Descriptor

An array bounds Descriptor is used to pass around all needed
information about an array and its associated dimensions.
Like VALU_DESCRIPTORs, they are simply 2-longword blocks,
but this might change.

!-----longword-----!
|-----|
| address of array |
|-----|
| length of array  |
|-----|

Such Descriptors must be accessed via the following
field names.
--

FIELD
  ARRAY_BNDS_SET =
  SET
    ARRAY_ADDRESS = [ 0,0,32,0 ], ! Beginning address of array.
    ARRAY_LENGTH  = [ 4,0,32,0 ], ! Size, in bytes, of array.
  TES;

!+
Declare an occurrence or REF to an array bounds
descriptor via the following macros.
!-

LITERAL
  ARRAY_BNDS_SIZE = 8; ! Each one is 2 longwords long.

MACRO
  ARRAY_BNDS_DESC = BLOCK[ ARRAY_BNDS_SIZE, BYTE ] FIELD( ARRAY_BNDS_SET ) %;
```


6569 0
6570 0
6571 0
6572 0
6573 0
6574 0
6575 0
6576 0
6577 0
6578 0
6579 0
6580 0
6581 0
6582 0
6583 0
6584 0
6585 0
6586 0

LITERAL SL_ACCE_INIT = 0, ! See above. "SL" --> SAT/LVT
SL_ACCE_RECS = 1;
SL_ACCE_SORT = 2;
SL_ACCE_FREE = 3;

+ You declare an occurrence or REF of an SAT datum via
the macro, SAT_RECORD.
-

MACRO SAT_RECORD = BLOCK[] FIELD(sat\$fld_def) %;

```

6587 0
6588 0
6589 0
6590 0
6591 0
6592 0
6593 0
6594 0
6595 0
6596 0
6597 0
6598 0
6599 0
6600 0
6601 0
6602 0
6603 0
6604 0
6605 0
6606 0
6607 0
6608 0
6609 0
6610 0
6611 0
6612 0
6613 0
6614 0
6615 0
6616 0
6617 0
6618 0
6619 0
6620 0
6621 0
6622 0
6623 0
6624 0
6625 0
6626 0
6627 0
6628 0
6629 0
6630 0
6631 0
6632 0
6633 0
6634 0
6635 0
6636 0
6637 0
6638 0
6639 0
6640 0
6641 0
6642 0
6643 0

```

++

Now we bring in the GST definitions directly from STARLET.REQ.

The format of one of these concatenated GSD records is a single leading byte containing the value 1, indicating that the record is indeed a GSD record.

Each entry in the record has a fixed number of overhead bytes followed by a symbol name that is a variable number of bytes. The entries we are interested in processing are the global symbol definitions and entry point symbol and mask definitions. The other defined type, PSECT definition, is noted only because it must be successfully passed over. The format of each of these types is illustrated below:

Global symbol definition:

0	! GSD type 1 !	
1	! data type !	ignored for now
2	! flag bytes !	bit 1 set means that this is a definition. ignore bit 0.
4	! psect index !	ignored.
5	! value !	4 bytes
9	! symbol name !	stock counted character string.

The entry point symbol and mask definition entry is identical to the global symbol definition illustrated above, with the addition of a two byte field for the procedure's register save mask. This two byte field is located after the symbol value field (which is an entry point address).

0	! GSD type 2 !	
1	! data type !	ignored for now
2	! flag bytes !	not relevant for entry point def.
4	! psect index !	ignored
5	! value !	4 bytes


```

6644 0
6645 0
6646 0
6647 0
6648 0
6649 0
6650 0
6651 0
6652 0
6653 0
6654 0
6655 0
6656 0
6657 0
6658 0
6659 0
6660 0
6661 0
6662 0
6663 0
6664 0
6665 0
6666 0
6667 0
6668 0
6669 0
6670 0
6671 0
6672 0
6673 0
6674 0
6675 0
6676 0
6677 0
6678 0
6679 0
6680 0
6681 0
6682 0
6683 0
6684 0
6685 0
6686 0
6687 0
6688 0
6689 0
6690 0
6691 0
6692 0
6693 0
6694 0
6695 0
6696 0
6697 0
6698 0
6699 0
6700 0

```

9	register	ignored,
10	save mask	2 bytes
11	symbol name	stock counted character string

The procedure definition with formal argument descriptions is identical to the "GSD type 2" definition above with the addition of fields that describe each formal argument. Following the symbol name there is one byte containing the minimum number of arguments allowed, and one byte containing the maximum number of arguments. These bytes are followed by a series of records of 2 - 257 bytes that describe each of the formal arguments (the number of these records is equal to the maximum number of arguments for the procedure.

0	! GSD type 3 !	
1	! data type !	ignored for now
2	! flag	bit 1 set indicates that this is
3	! bytes !	a definition. Bit 0 ignored.
4	! psect index !	ignored
5	! value !	4 bytes
9	! register	ignored,
10	! save mask !	2 bytes
11	! symbol	stock counted character
	! name !	string
	! min # of args !	1 byte
	! max # of args !	1 byte
	! formal arg #1	
	! description !	
	!	
	!	
	! formal arg #n	
	! description !	

Format of each formal argument description.

[illegible]


```

6758 0      GST_G_NAME_ADDR = [10,0, 8,0 ],
6759 00
6760 00
6761 00
6762 00
6763 00      GST_E_NAME_CS   = [11,0, 8,0 ],
6764 00
6765 00
6766 00
6767 00      GST_E_NAME_ADDR = [12,0, 8,0 ],
6768 00
6769 00
6770 00      GST_P_MAX_ARG   = [ 1,0, 8,0 ],
6771 00      GST_P_REM_CNT   = [ 1,0, 8,0 ],
6772 00
6773 00      TES;
6774 00
6775 00      +
6776 00      You declare an occurrence or REF of a GST datum via:
6777 00      -
6778 00
6779 00      LITERAL
6780 00      GST_RECORD_SIZE = 43;    ! Each GST record is at most 43 bytes long.
6781 00
6782 00      MACRO
6783 00      GST_RECORD = BLOCK[ GST_RECORD_SIZE, BYTE] FIELD( GST_FIELD_SET ) %;
6784 00
6785 00
6786 00      LITERAL
6787 00      GST_PSECT_OVERHEAD = 9,    ! Minimum number of bytes in psect def
6788 00      GST_GLOBAL_OVERHEAD = 10,   ! Minimum number of bytes in a global entry
6789 00      GST_ENTRY_OVERHEAD = 12,    ! Minimum number of bytes in entry
6790 00                                   ! point symbol and mask definition
6791 00      GST_ARGDSC_OVERHEAD = 2,    ! Minimum size of formal argument descriptor
6792 00      GST_MINMAX_OVERHEAD = 2,    ! Size of min. and max. overhead in GST
6793 00      GST_RECORD_TYPE = 0,        ! Type of GST record
6794 00      GST_TYPE = 1;               ! Record Type is GST
6795 00
6796 00      +
6797 00      The GST record types are defined as:
6798 00      -
6799 00
6800 00      LITERAL
6801 00      ! GST types:
6802 00
6803 00      GST_LOWEST      = 1,         ! We don't support global PSECTs now.
6804 00
6805 00      GST_PSECT_DEFN  = 0,         ! P-SECT record.
6806 00      GST_GLOBAL_DEFN = 1,         ! A global symbol definition record.
6807 00      GST_ENTRY_DEFN  = 2,         ! An entry point definition.
6808 00      GST_PROC_DEFN   = 3,         ! A procedure with formal argument desc.
6809 00
6810 00      GST_HIGHEST    = 3;         ! Highest one we support.

```

! The name string itself. An undotted
 ! reference is the address of the name,
 ! a dotted one is the 1st character.

! The entry name is a counted string.
 ! A dotted reference to this field
 ! picks up the count, an undotted
 ! one addresses the counted string.
 ! The name string itself. An undotted
 ! reference is the address of the name,
 ! a dotted one is the 1st character.
 ! Maximum number of formal arguments.
 ! Remaining byte count of argument
 ! descriptor.

! Minimum number of bytes in psect def
 ! Minimum number of bytes in a global entry
 ! Minimum number of bytes in entry
 ! point symbol and mask definition
 ! Minimum size of formal argument descriptor
 ! Size of min. and max. overhead in GST
 ! Type of GST record
 ! Record Type is GST

```

6811 0
6812 0
6813 0
6814 0
6815 0
6816 0
6817 0
6818 0
6819 0
6820 0
6821 0
6822 0
6823 0
6824 0
6825 0
6826 0
6827 0
6828 0
6829 0
6830 0
6831 0
6832 0
6833 0
6834 0
6835 0
6836 0
6837 0
6838 0
6839 0
6840 0
6841 0
6842 0
6843 0
6844 0
6845 0
6846 0
6847 0
6848 0
6849 0
6850 0
6851 0
6852 0
6853 0
6854 0
6855 0
6856 0
6857 0
6858 0
6859 0
6860 0
6861 0
6862 0
6863 0
6864 0
6865 0
6866 0
6867 0

```

!++
BLISS uses 'non-standard' DST records to encode
most of its local symbol information. These records
are like most DST records except that the TYPE
information is variable-sized.
--

FIELD
SET BLZ_FIELD_SET =
BLZ_SIZE = [0,0, 8,0], ! First byte is record size in bytes.
! The next byte contains DSC\$K_DTYPE_Z, or we
! wouldn't be applying this structure to a given
! DST record.
BLZ_TYP_SIZ = [2,0, 8,0], ! Type info takes up this
! many bytes.
BLZ_TYPE = [3,0, 8,0], ! Which type of type Zero
! this corresponds to.
BLZ_ACCESS = [4,0, 8,0], ! Access field.
! Sub fields of _ACCESS are offset from beginning
! of the BLZ record.
BLZ_ACCES_TYPE = [4,0, 2,0], ! Type of access,
BLZ_ACCES_BASD = [4,2, 2,0], ! based or not,
BLZ_ACCES_BREG = [4,4, 4,0], ! associated register.
BLZ_STRUCT = [5,0,3,0], ! Type of STRUCTURE reference.
BLZ_REF = [5,7,1,0], ! Indicates whether symbol has REF attribute
! **** The following only work when BLZ_TYP_SIZ is 3.
BLZ_VALUE = [6,0,32,0], ! DST VALUE field.
BLZ_NAME_CS = [10,0, 8,0], ! The symbol name is a counted string.
! A dotted reference to this field
! picks up the count, an undotted
! one addresses the counted string.
BLZ_NAME_ADDR = [11,0, 8,0], ! The name string itself. An undotted
! reference is the address of the name,
! a dotted one is the 1st character.
! The following fields are in the variable length part of the DST record.
! They should only be applied once the structure type has been determined
! or errors could result.
U_ALLOC_STRUC = [6, 0, 32, 0], ! no of units alloc (except BLOCKVECTOR)
U_ALLOC_BVEC = [10, 0, 32, 0], ! no of units alloc for BLOCKVECTOR
SIGN_EXT_VEC = [10, 7, 1, 0], ! sign ext field for VECTOR
UNIT_SIZE_BLOCK = [10, 0, 8, 0], ! unit size for BLOCK
UNIT_SIZE_BVEC = [14, 0, 8, 0], ! unit size for BLOCKVECTOR
UNIT_SIZE_VEC = [10, 0, 4, 0], ! unit size for VECTOR
TES;
!+
! You declare a REF to a BLZ_DST datum via:


```

: 6868 0  !-
: 6869 0
: 6870 0  LITERAL
: 6871 0    BLZ_REC_SIZ      = 54;    ! Each DST record is at most 54 bytes long.
: 6872 0
: 6873 0  MACRO
: 6874 0    BLZ_RECORD = BLOCK[ BLZ_REC_SIZ, BYTE] FIELD( BLZ_FIELD_SET ) %;
: 6875 0
: 6876 0
: 6877 0  !+
: 6878 0    ! The type zero sub types,
: 6879 0    ! as defined in CP0021.MEM,
: 6880 0    ! must be within the following
: 6881 0    ! range.
: 6882 0  !-
: 6883 0
: 6884 0  LITERAL
: 6885 0
: 6886 0    ! Type Zero Sub-Types:
: 6887 0
: 6888 0    BLZ_LOWEST      = 1,      ! Lowest variable type we support.
: 6889 0
: 6890 0    BLISS_Z_FORMAL  = 1,      ! Description of a ROUTINE formal.
: 6891 0    BLISS_Z_SYMBOL  = 2,      ! A BLISS LOCAL symbol.
: 6892 0
: 6893 0    BLZ_HIGHEST     = 2;      ! Highest variable type we support.
: 6894 0

```

```

6895 0
6896 0
6897 0
6898 0
6899 0
6900 0
6901 0
6902 0
6903 0
6904 0
6905 0
6906 0
6907 0
6908 0
6909 0
6910 0
6911 0
6912 0
6913 0
6914 0
6915 0
6916 0
6917 0
6918 0
6919 0
6920 0
6921 0
6922 0
6923 0
6924 0
6925 0
6926 0
6927 0

```

Dummy descriptors

Whenever a type DSC\$K_DTYPE_CAD item is being accessed by descriptor we must build a dummy one in free storage and put the current address in it. The address of this dummy descriptor is then used by the rest of the debugger in referencing the item. To be able to free up all this space, the items are kept on a linked list. At the end of command processing this list is walked down and all the space is returned to the free storage manager.

Each entry on the linked list is a 3 longword vector

pointer to next item on the linked list
pointer to the dummy descriptor
size in bytes of the dummy descriptor

The global variable DBG\$GL_DLISHEAD if nonzero points to the first item on the list.

LITERAL

```

DLIS_ENTRY = 3 ;      ! Size in longwords of entry on list
DLIS_LINK = 0 ;      ! Pointer to next item on list
DLIS_POINTER = 1 ;    ! Pointer to dummy descriptor
DLIS_DESCSIZE = 2 ;   ! Size in bytes of descriptor

```


H 8
15-Sep-1984 22:59:57
15-Sep-1984 22:43:00

VAX-11 Bliss-32 V4.0-742
_S255\$DUA28:[DEBUG.SRC]DBGLIB.REQ;1 Page 164
(98)

; 6928 0 ! END OF DBGLIB.REQ

Library Statistics

File	----- Total	Symbols Loaded	----- Percent	Pages Mapped	Processing Time
_S255\$DUA28:[DEBUG.OBJ]STRUCDEF.L32;1	32	18	56	7	00:00.1

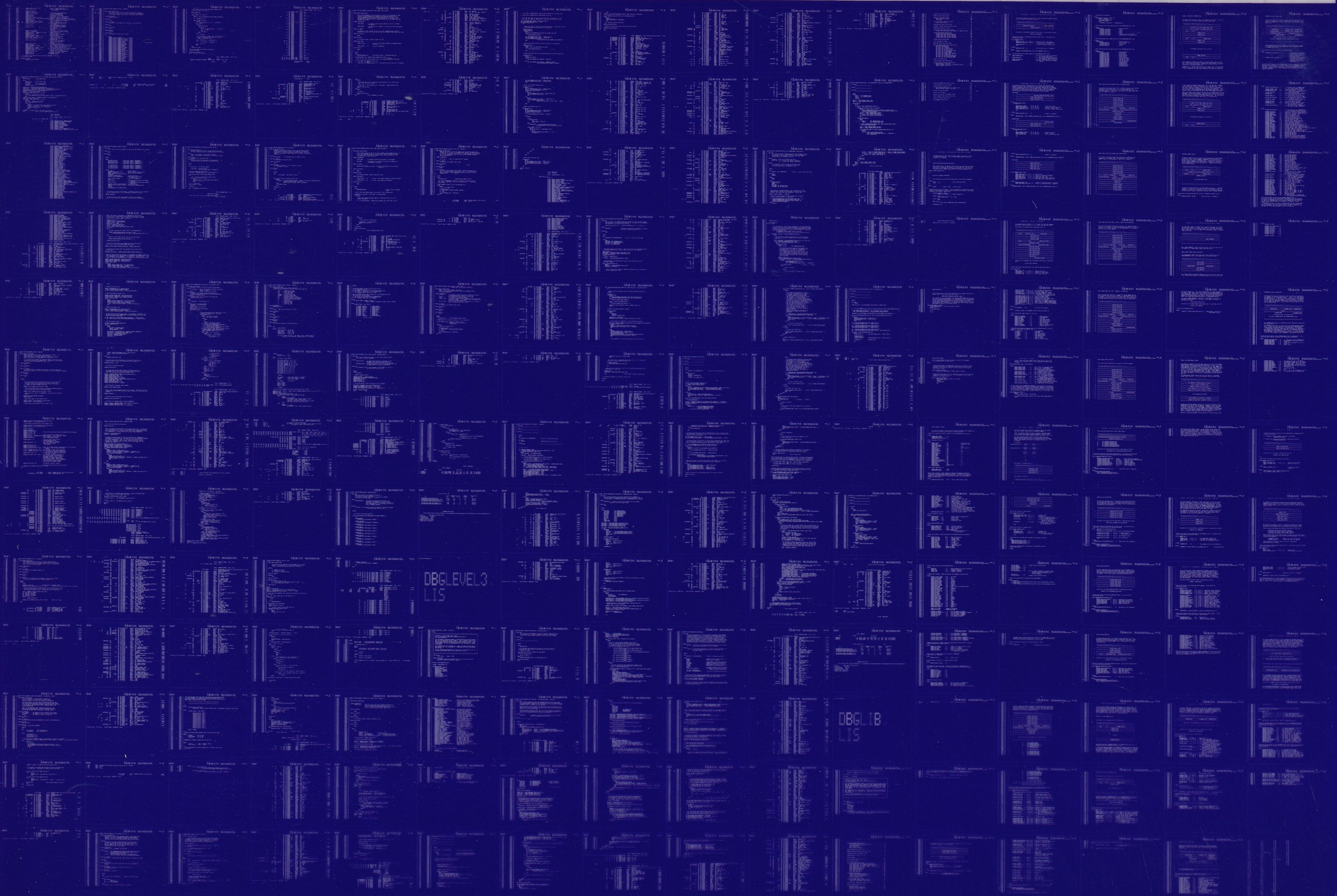
COMMAND QUALIFIERS

; BLISS/LIBRARY=LIB\$:DBGLIB.L32/LIST=LISS\$:DBGLIB.LIS SRC\$:DBGLIB.REQ

; Run Time: 00:48.9
; Elapsed Time: 01:07.9
; Lines/CPU Min: 8504
; Lexemes/CPU-Min: 21240
; Memory Used: 298 pages
; Library Precompilation Complete

0085 AH-BT13A-SE
VAX/VMS V4.0

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY



0086 AH-BT13A-SE
VAX/VMS V4.0

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY

